Java Database Connectivity

SUBJECT:
Advanced Java Programming

TOPIC:
Java Database Connectivity

Outline

- JDBC Architecture
- Types of JDBC Drivers
- Creating simple JDBC Application
  - Classes and Interfaces
- Types of Statements
  - Statement Interface, PreparedStatement, CallableStatement
- Exploring ResultSet Operations
  - ResultSet, ResultSetMetadata, DatabaseMetadata
- Batch Updates in JDBC
- Using Rowsets Objects
- Managing DatabaseTransaction
- Creating CRUD Application

What is JDBC ??

- The JDBC (Java Database Connectivity) interface is a pure Java API used to execute SQL Statements.

- JDBC is a Java database connectivity technology (Java Standard Edition platform) from Oracle Corporation.

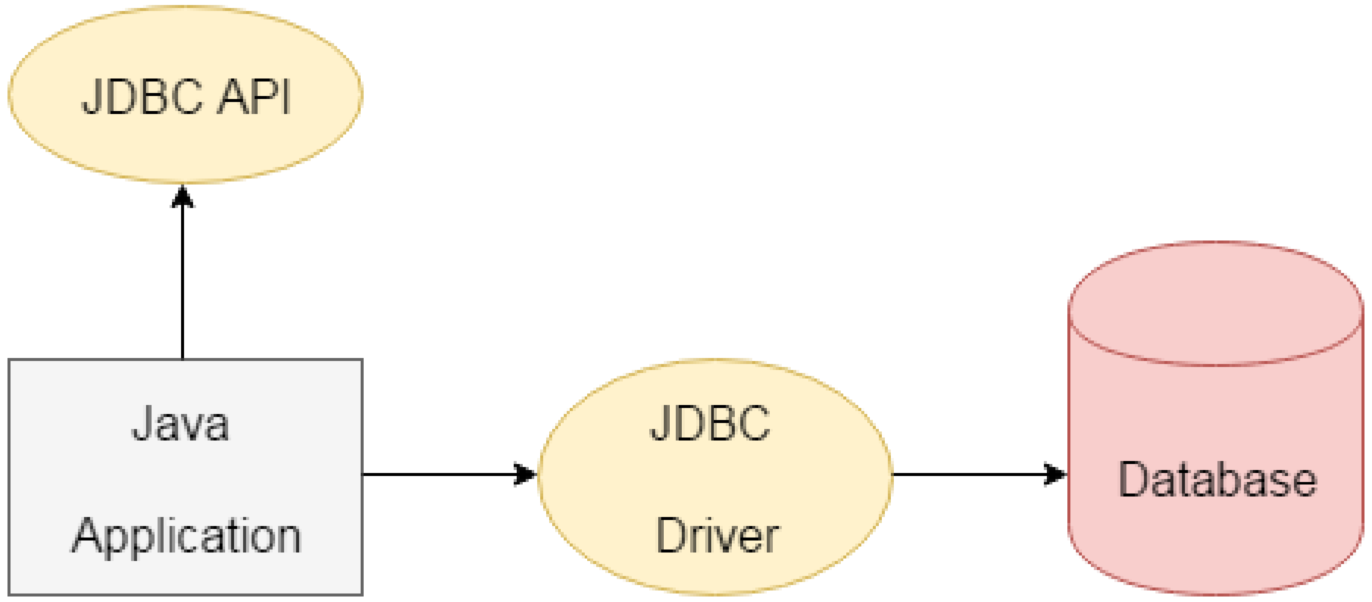
- This technology is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database.
- It provides a set of classes & interfaces that can be used by developers to write a database applications in Java.

JDBC Goals

- SQL-Level
- 100% Pure Java
- Keep it simple
- High-performance
- Use of existing database technology
- Use multiple methods to express multiple functionality

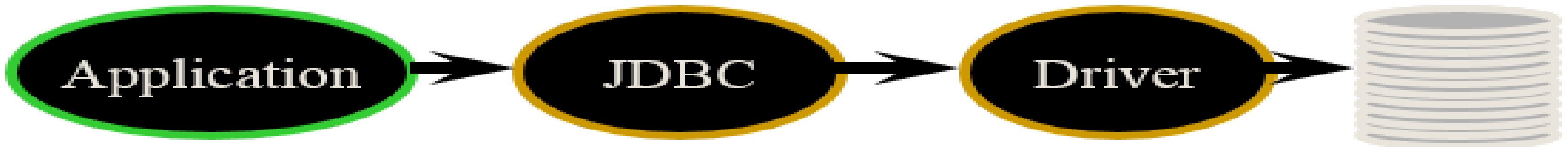
JDBC : INTRODUCTION

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

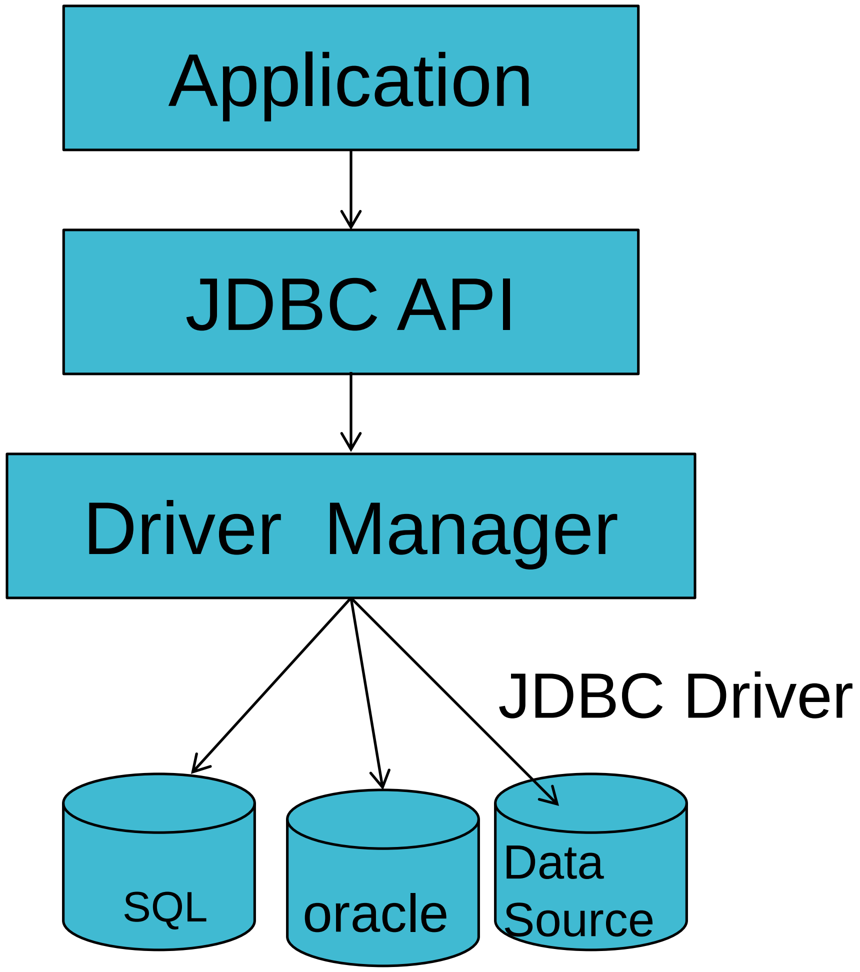


JDBC : ARCHITECTURE

- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines without changing any application code



JDBC Architecture



JDBC Architecture

- The JDBC architecture consist of two major components : **JDBC API** and **JDBC driver type.**
- The **JDBC API is a set of classes, interface** used to establish connection with data source. The JDBC API is defined in the **java.sql** and **javax.sql** package.
- We use following core JDBC classes and interfaces that belong to java.sql package:
- **DriverManager :** When java application needs connection to the Db it invokes the DriverManager class. This class **load JDBC drivers in the memory**.
- **Connection :** This is an **interface** which represents connectivity with the data source.

JDBC

Some basic components

Components of JDBC

1. JDBC API

- Using JDBC API ,front end java applications can execute query and fetch data from connected database.
- JDBC API can also connect with multiple application with same database

OR

- Same application with multiple Databases which can be reside in different computers(distributed environment).

Components of JDBC

2. JDBC Driver Manager

- ' Driver Manager' of JDBC interface defines 'objects' which is used to connect java application with 'JDBC Driver'.
- 'Driver Manager' manages the all the 'JDBC Driver' loaded in client's system.
- 'Driver Manager' load the most appropriate driver among all the Drivers for creating a connection.

Components of JDBC

3. JDBC Test Suite

- JDBC Test Suite is used to check compatibility of a JDBC driver with platform.

- It also check whether a Driver follow all the standard and requirements of Environment .

Components of JDBC

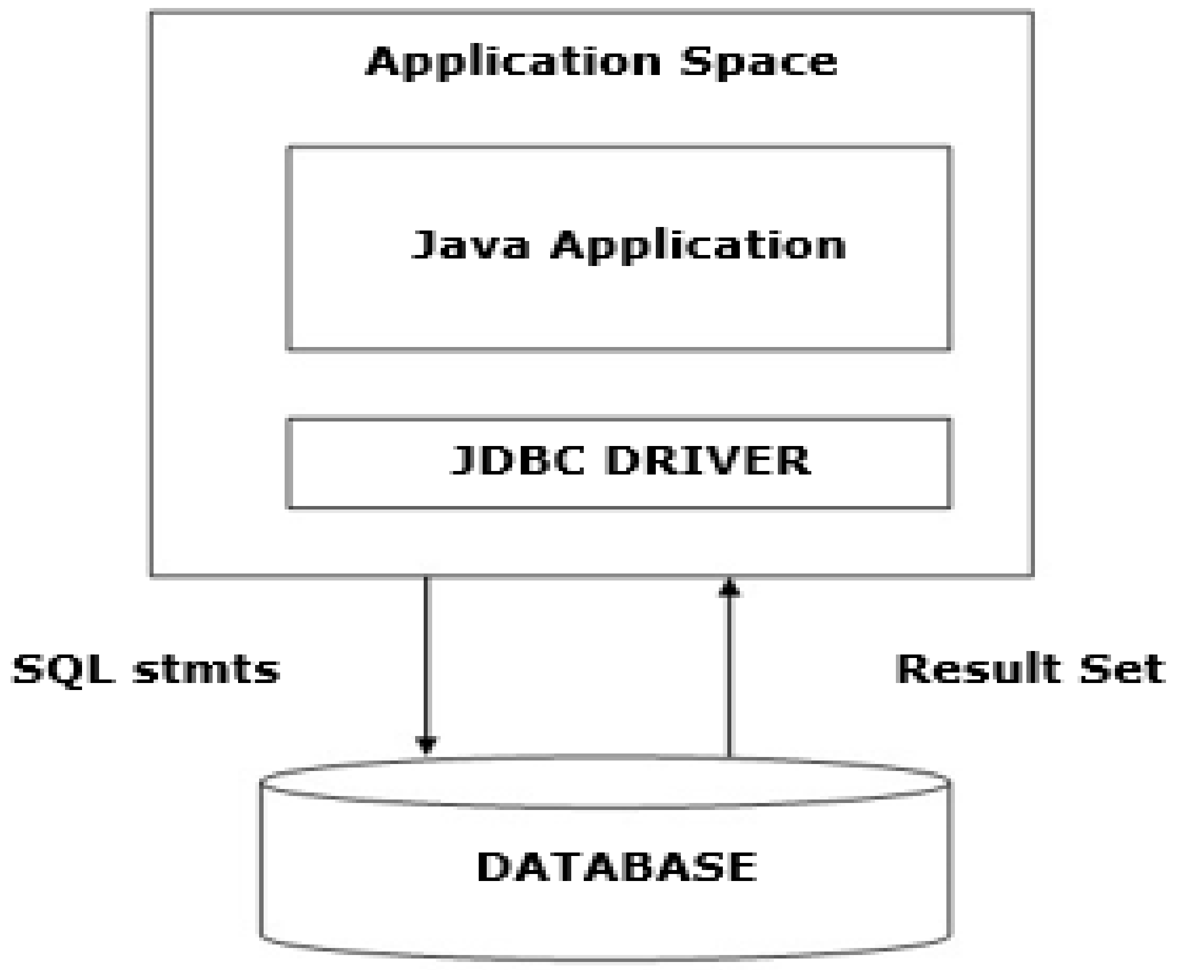
4. JDBC-ODBC BRIDGE

- The ODBC(Open database connectivity) bridge or drivers should be installed on work site for proper working of this component .The JDBC Driver contact to the ODBC Driver for connection to the database.
- The ODBC Driver is already installed or come as default driver in windows for pcs .In Windows 'Datasource' name can be create using control panel >administrative tools>Data Sources (ODBC).
- AFTER Creating 'datasource' ,connectivity of 'datasource' to the 'database' can be check .Using this "data source", you can connect JDBC to ODBC.

JDBC Drivers

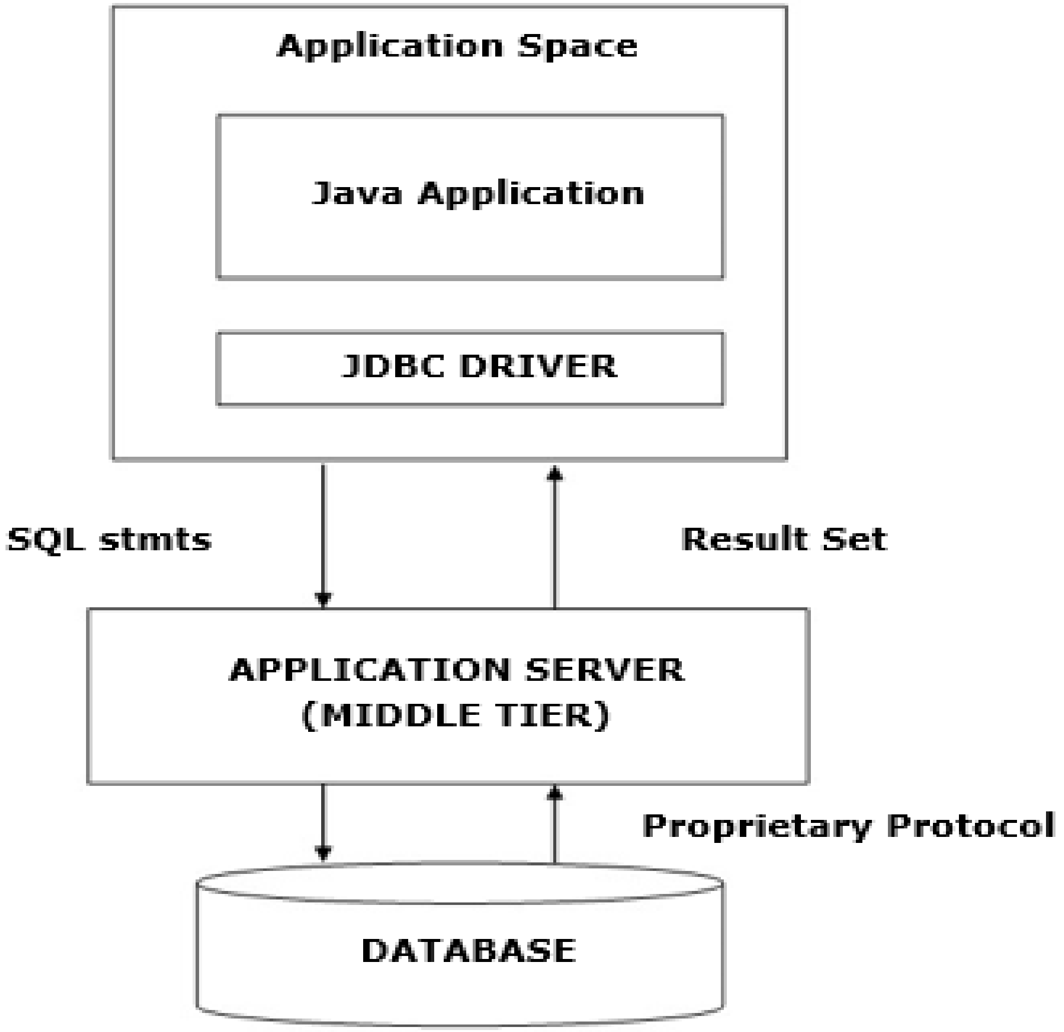
Two Tier Access Models

- Your Java Application talks directly to the DB. This is accomplished through the Use of JDBC DRIVER, which sends commands directly to DB. The results are sent back from the DB directly to the application.



Three Tier Access Models

- Your JDBC DRIVER sends commands to a Middle Tier, which in turn sends commands to the DB. The results of these commands are sent back to middle tier, which sends them back to the application.



JDBC Driver Types

- A JDBC driver is a software component enabling a Java application to interact with a database.
- To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database.
- The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

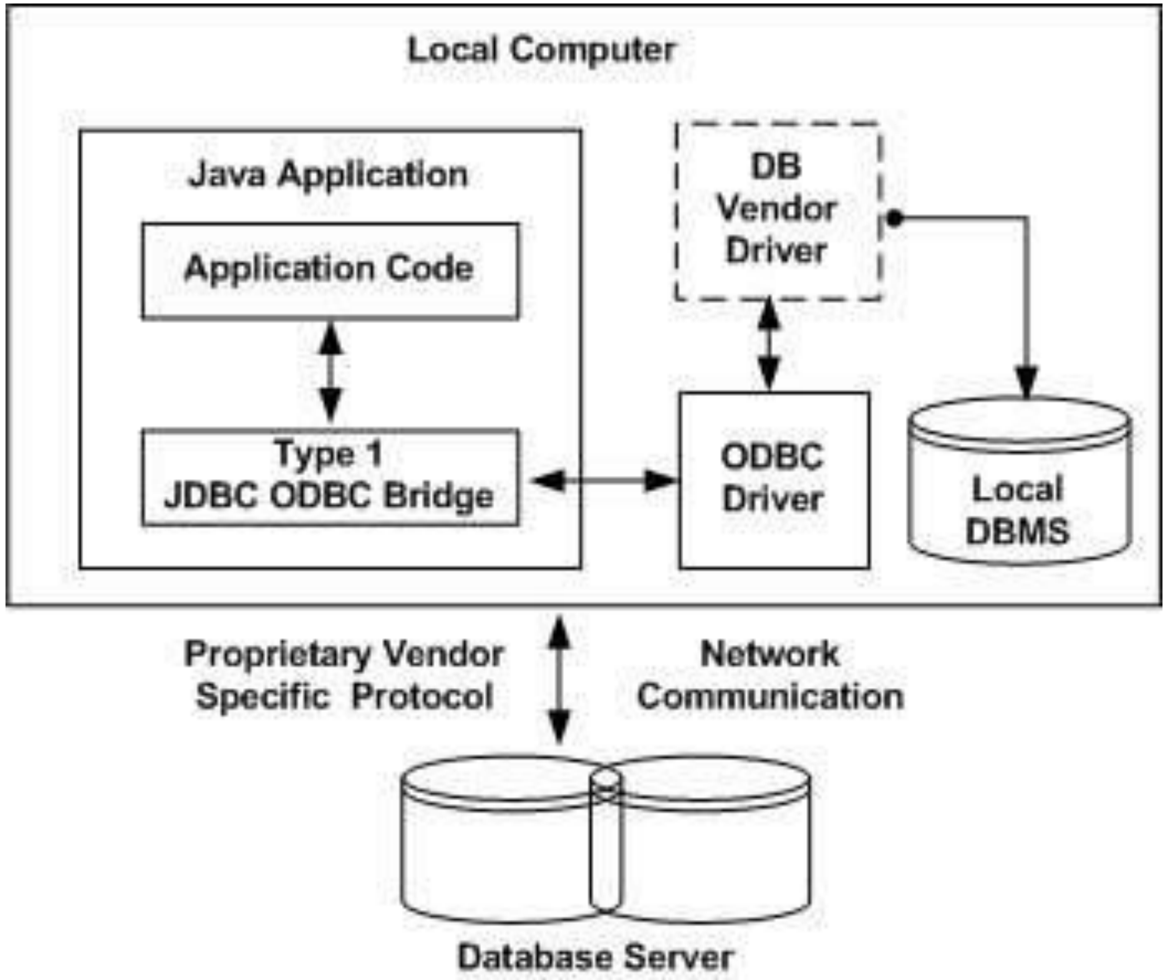
4 Types of JDBC Drivers

JDBC drivers are divided into four types or levels:

- **Type 1:** JDBC-ODBC Bridge driver (Bridge)
- **Type 2:** Native-API / partly Java driver (Native)
  - Native-API driver (partially java driver)
- **Type 3:** All Java / Network-protocol driver (Middleware)
  - Network Protocol driver (fully java driver)
- **Type 4:** All Java / Native-protocol driver (Pure)
  - Thin driver (fully java driver)

Type 1: JDBC-ODBC bridge driver

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



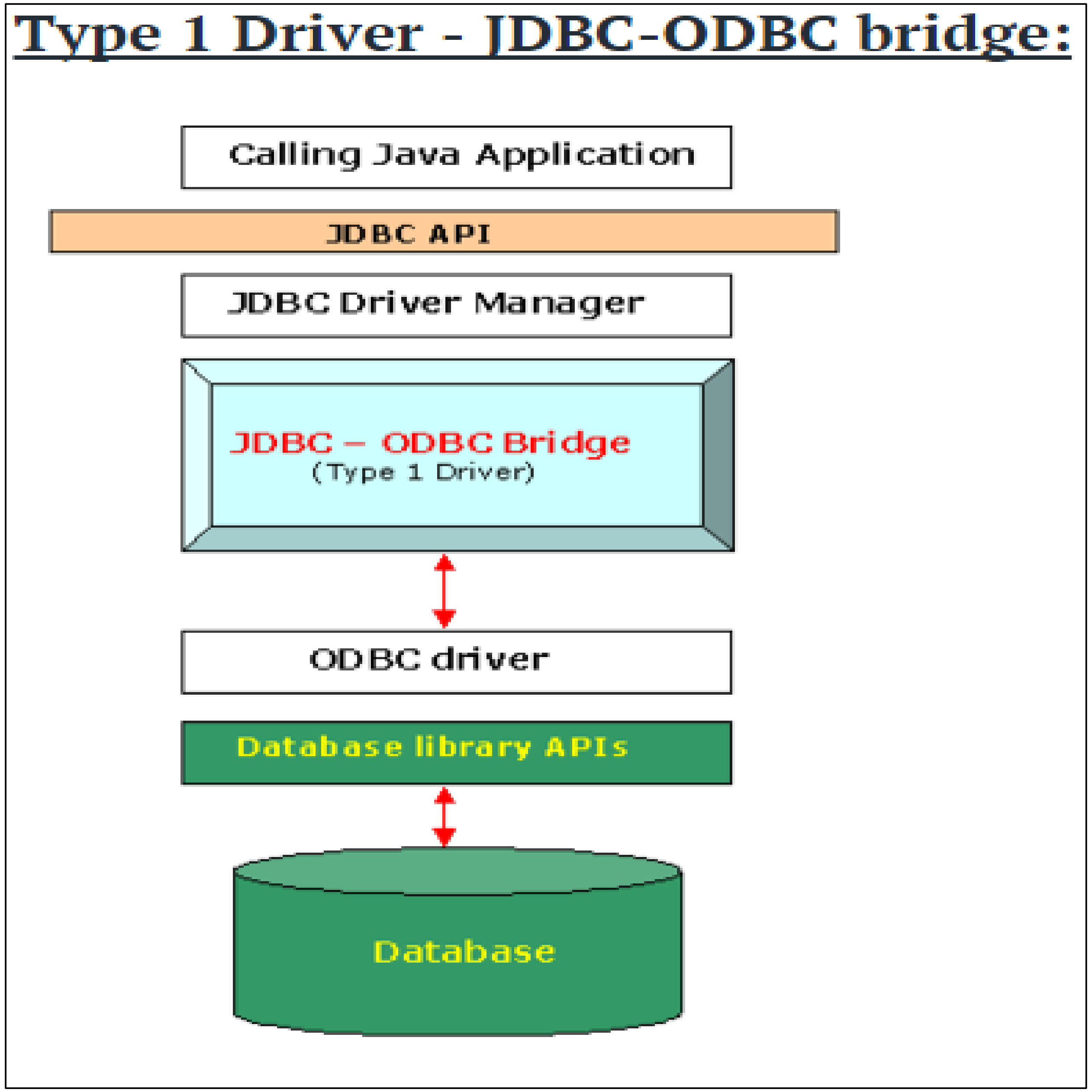
Type 1: JDBC-ODBC bridge driver

- The JDBC type 1 driver, also known as the JDBC-ODBC bridge, is a database driver implementation that employs the ODBC driver to connect to the database.
- The driver converts JDBC method calls into ODBC function calls.
- The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon.
- Also, use of this driver leads to other installation dependencies; for example, ODBC must be installed on the computer having the driver and the database must support an ODBC driver.
- The use of this driver is discouraged if the alternative of a pure-Java driver is available. This technology isn't suitable for a high-transaction environment.

Type 1: JDBC-ODBC bridge driver

- Type 1 drivers also don't support the complete Java command set and are limited by the functionality of the ODBC driver.
- There is a JDBC-ODBC Bridge driver:
  - sun.jdbc.odbc.JdbcOdbcDriver.
- It may sometimes be the case that more than one JDBC driver is capable of connecting to a given URL.
- For example, when connecting to a given remote database, it might be possible to use a JDBC-ODBC bridge driver, a JDBC-to-generic-network-protocol driver, or a driver supplied by the database vendor.
- In such cases, the order in which the drivers are tested is significant because the DriverManager will use the first driver it finds that can successfully connect to the given URL.

Type 1: JDBC-ODBC bridge driver



Type 1: JDBC-ODBC bridge driver

Advantages:

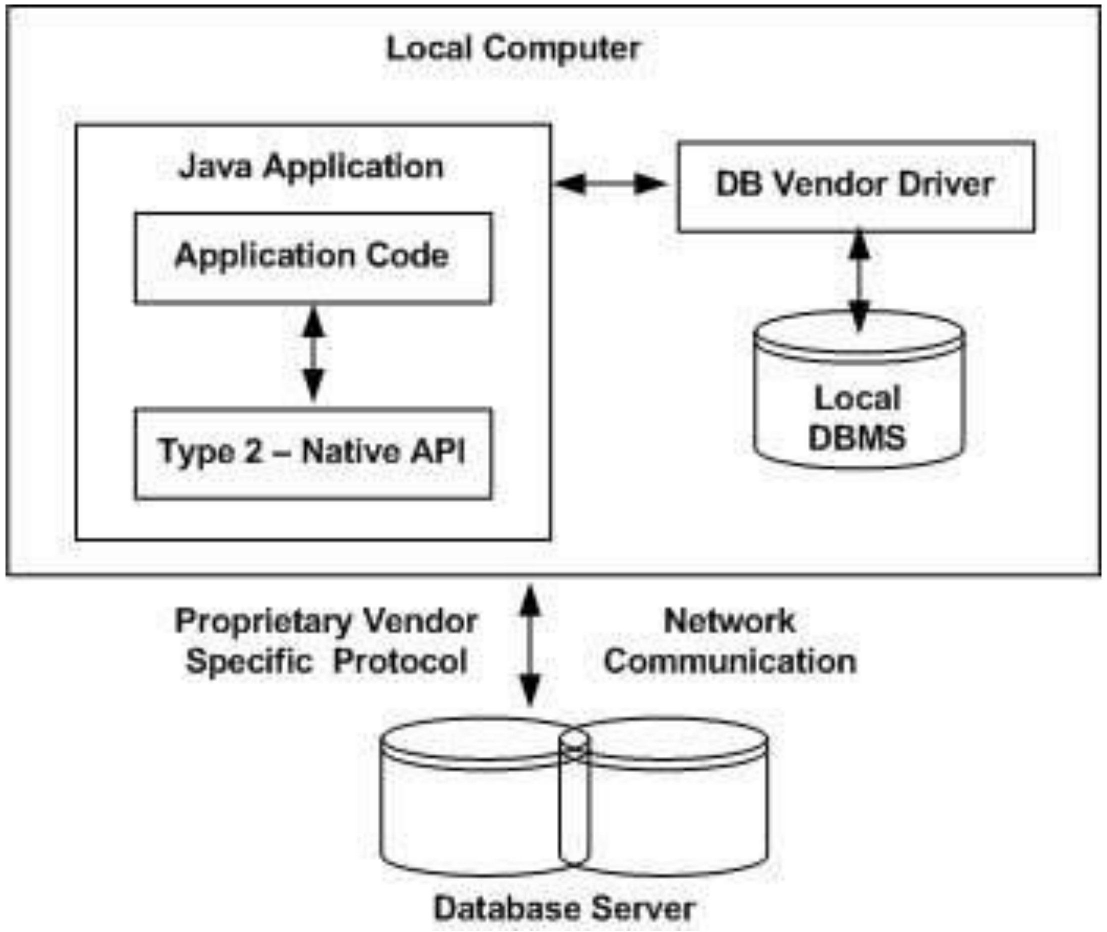
- Almost any database for which ODBC driver is installed, can be accessed.

Disadvantages:

- Performance overhead since the calls have to go through the jdbc overhead bridge to the ODBC driver, then to the native db connectivity interface (thus may be slower than other types of drivers).
- The ODBC driver needs to be installed on the client machine.
- Not suitable for applets, because the ODBC driver needs to be installed on the client.

Type 2: Native-API driver - Net pure Java

- In a Type 2 driver, **JDBC API calls are converted into native C/C++ API calls, which are unique to the database.** These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some **speed increase with a Type 2 driver**, because it eliminates ODBC's overhead.



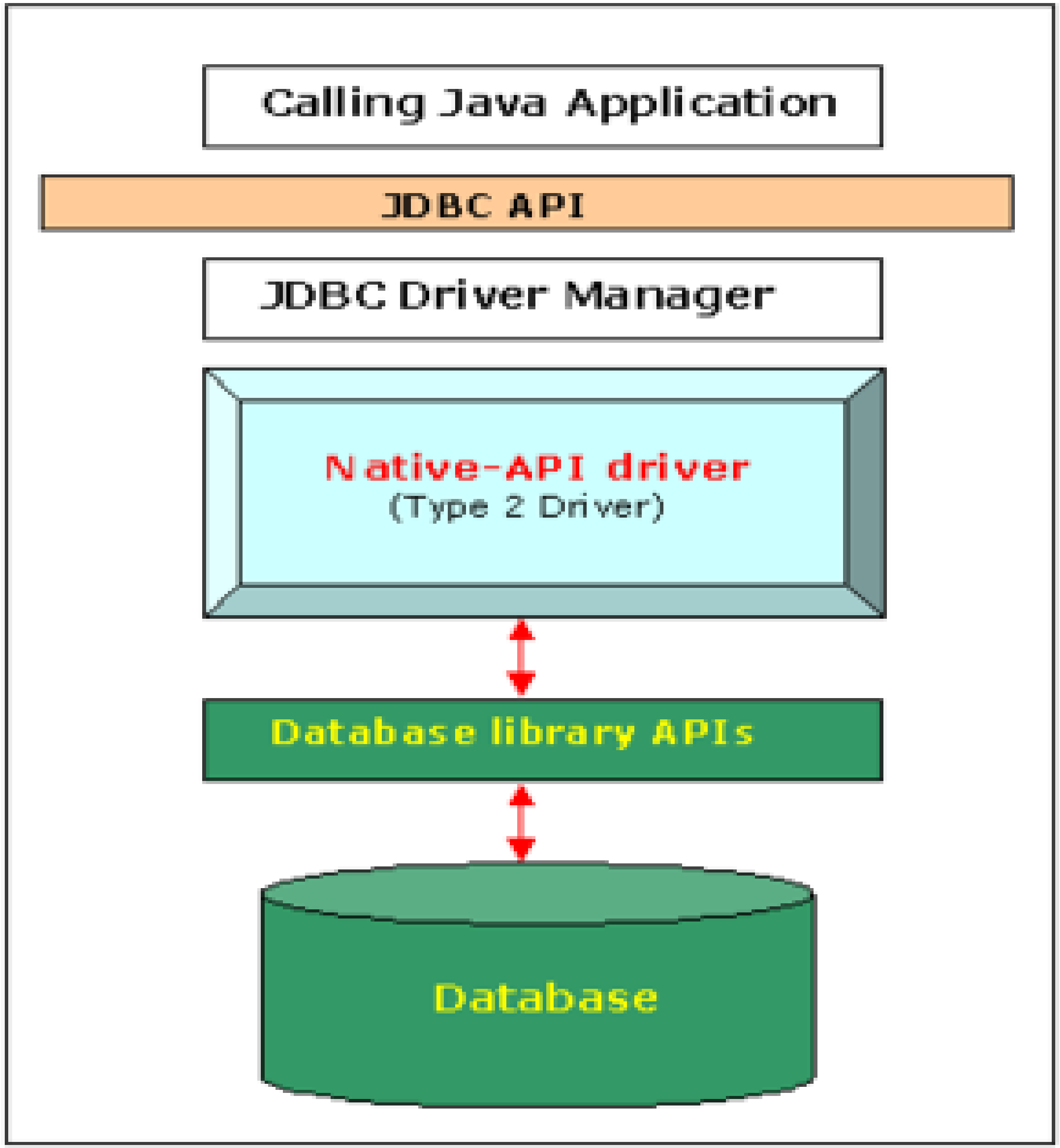
Type 2: Native-API driver - Net pure Java

- The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

- The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database.

- Example: Oracle will have oracle native api.

Type 2: Native-API driver - Net pure Java



Type 2: Native-API driver - Net pure Java

Advantages:

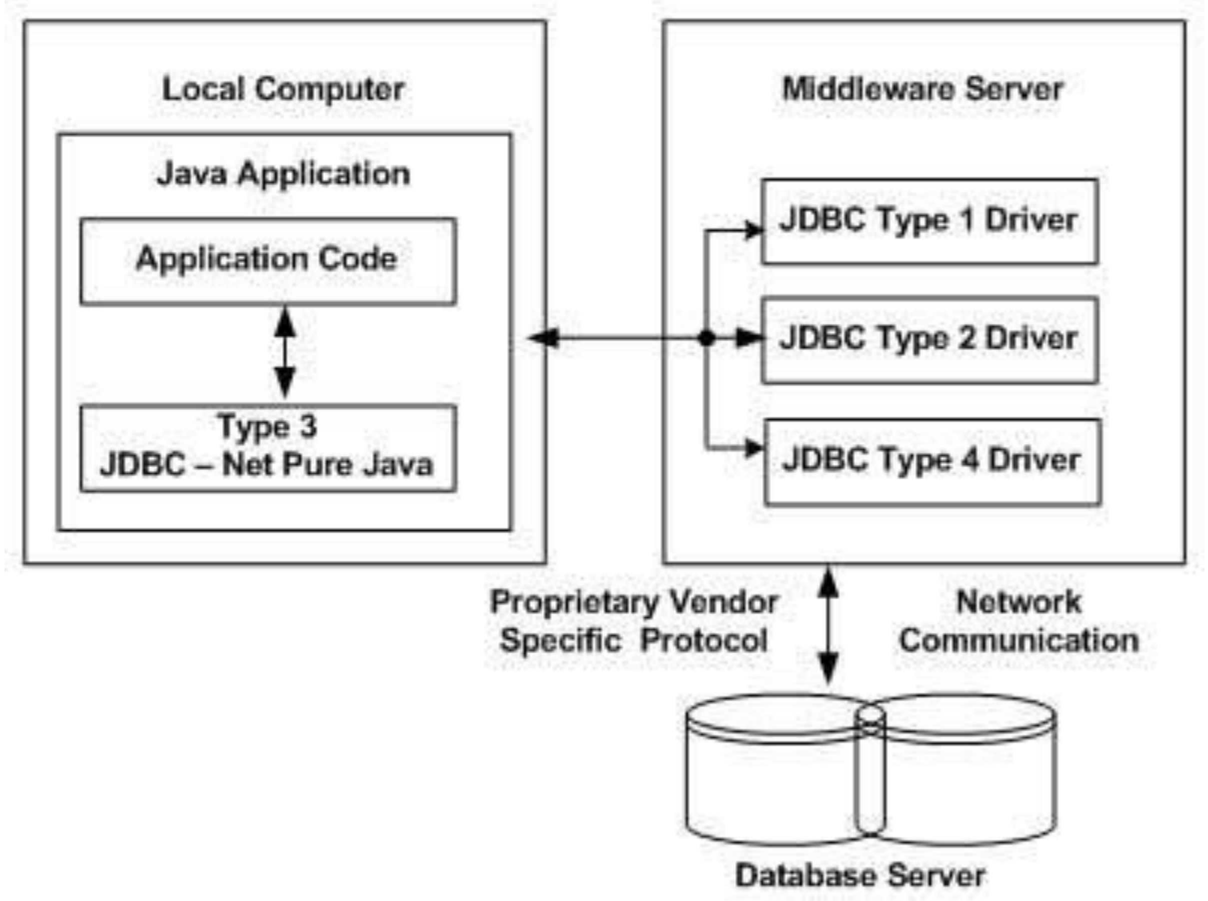
- As there is no implementation of Jdbc-Odbc bridge, its considerably faster than a type 1 driver.

Disadvantages:

- The vendor client library needs to be installed on the client machine.
- Not all databases have a client side library.
- This driver is platform dependent.
- This driver supports all java applications except Applets.

Type 3: Network Protocol driver

- In a Type 3 driver, **a three-tier approach is used to access databases.** The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the **middleware** application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



Type 3: Network Protocol driver

- The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database.
- The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- This differs from the type 4 driver in that the protocol conversion logic resides not at the client, but in the middle-tier.

Type 3: Network Protocol driver

- Like type 4 drivers, the type 3 driver is written entirely in Java. The same driver can be used for multiple databases.
- It depends on the number of databases the middleware has been configured to support.
- The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware.
- Use of the middleware provides additional advantages of security and firewall access.

Type 3: Network Protocol driver

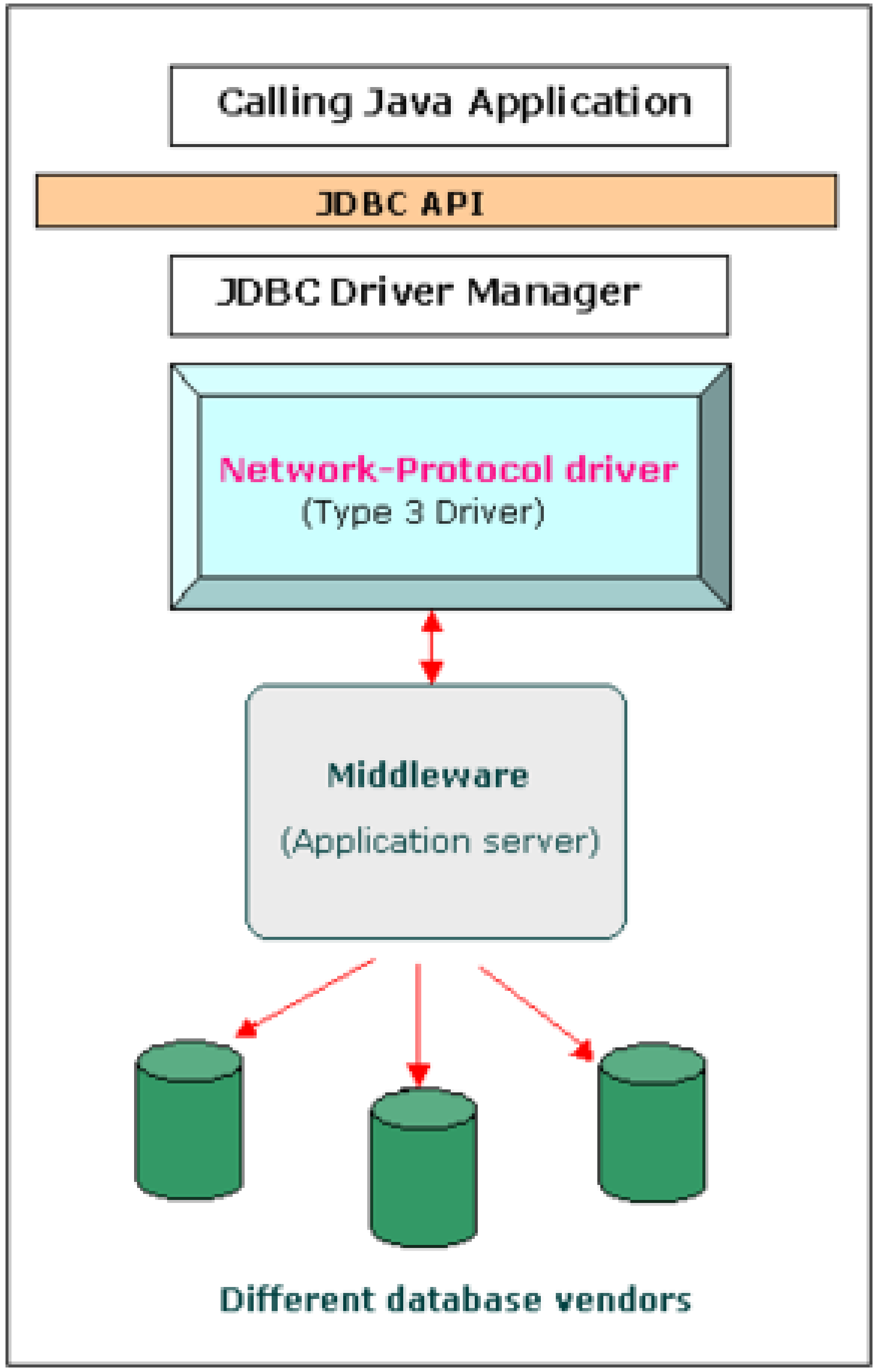
Functions:

- Sends JDBC API calls to a middle-tier net server that translates the calls into the DBMS-specific network protocol. The translated calls are then sent to a particular DBMS.

- Follows a three tier communication approach. Can interface to multiple databases - Not vendor specific.

- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.

Type 3: Network Protocol driver



Type 3: Network Protocol driver

Advantages:

- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client. The client need not be changed for a new database.
- The middleware server (which can be a full Application server) can provide typical middleware services like caching (of connections, query results, etc.), load balancing, logging, and auditing.
- A single driver can handle any database, provided the middleware supports it.

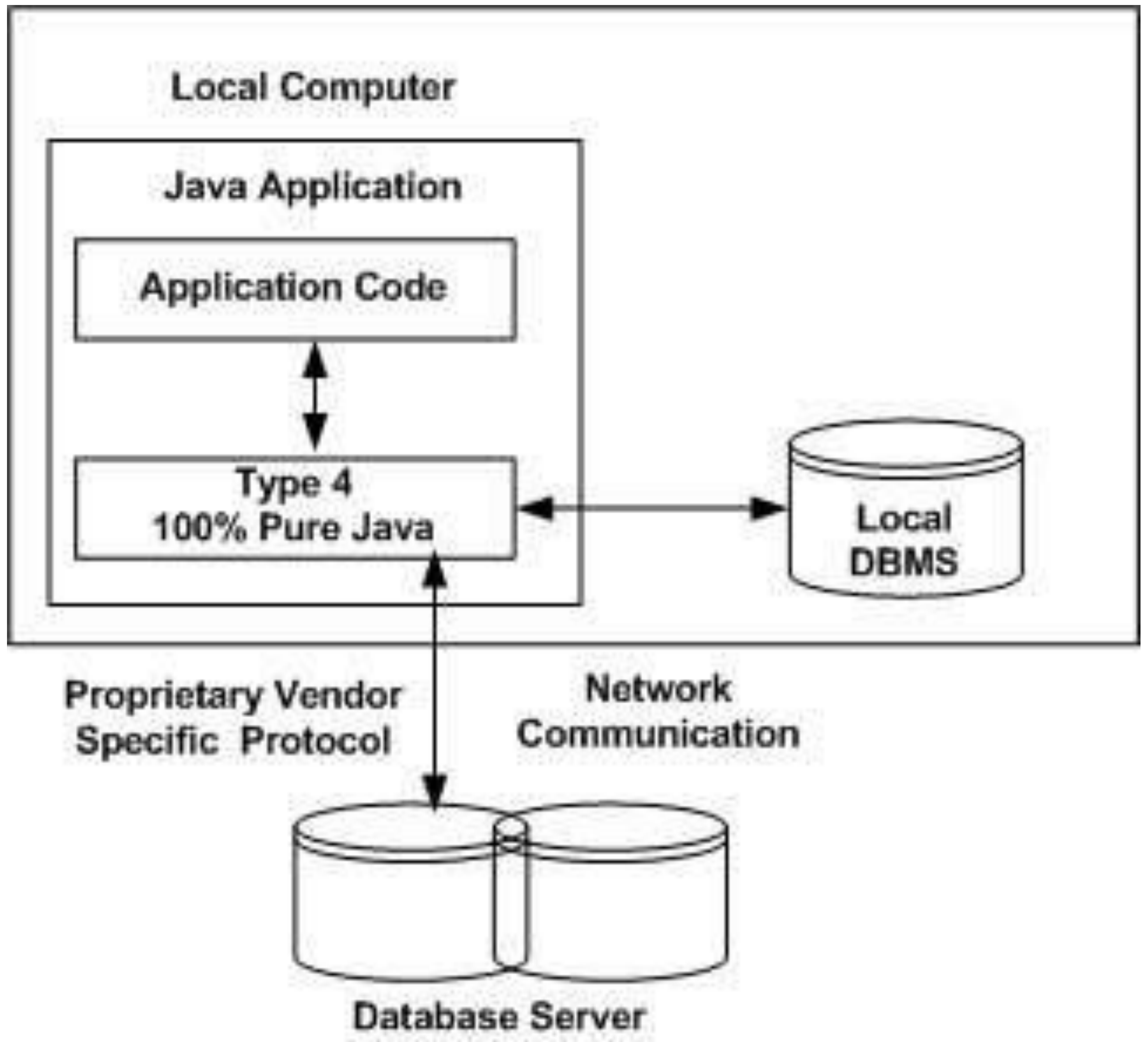
Type 3: Network Protocol driver

Disadvantages:

- Requires database-specific coding to be done in the middle tier.
- The middleware layer added may result in additional latency, but is typically overcome by using better middleware services.

Type 4: Thin driver - 100% Pure Java (Native-Protocol)

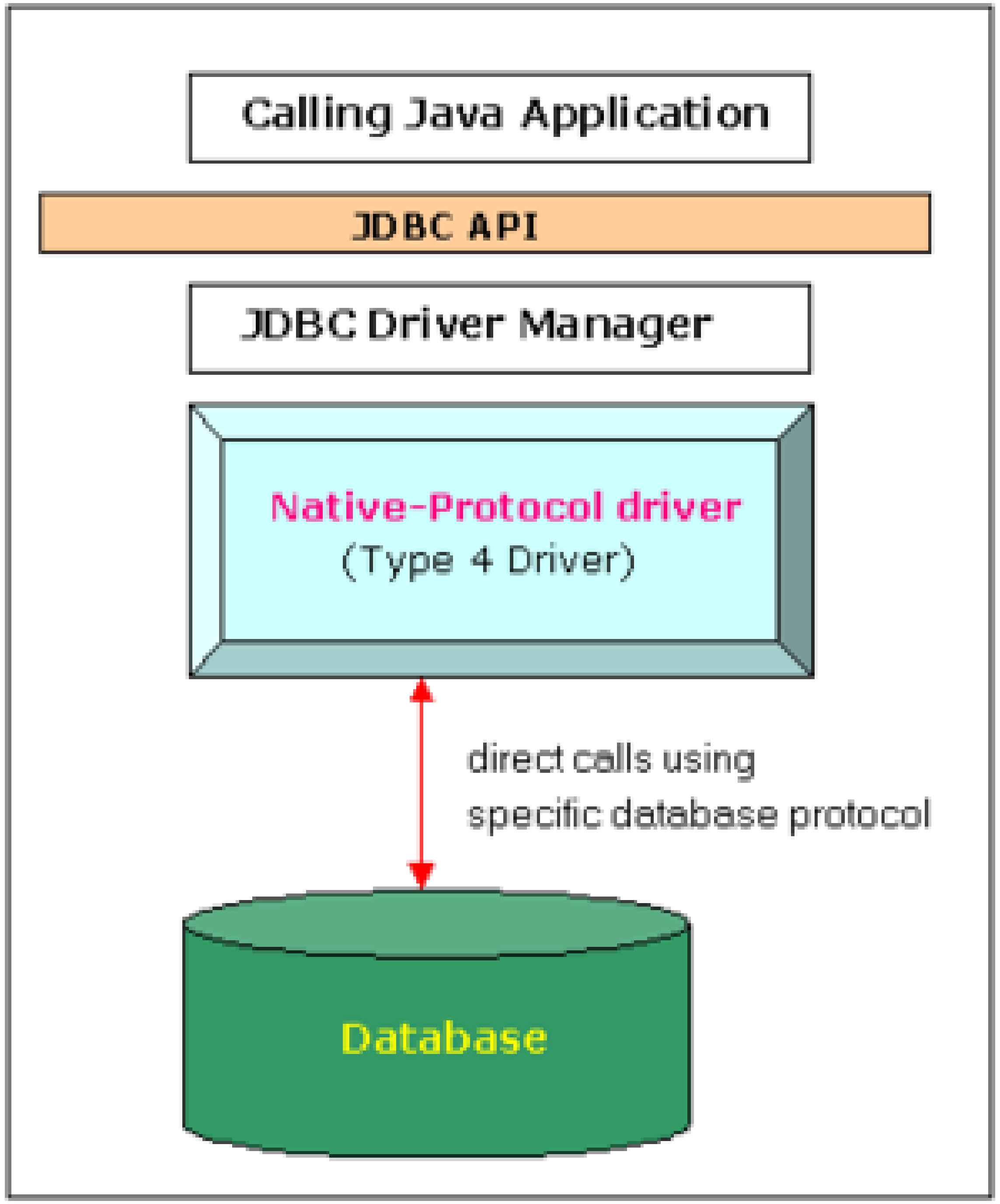
- In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through **socket connection**. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



Type 4: Thin driver - 100% Pure Java (Native-Protocol)

- The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.
- Written completely in Java, type 4 drivers are thus platform independent.
- This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls.
- Unlike the type 3 drivers, it does not need associated software to work.

Type 4: Thin driver - 100% Pure Java (Native-Protocol)



Type 4: Thin driver - 100% Pure Java (Native-Protocol)

Advantages:

- Completely implemented in Java to achieve platform independence.
- These drivers don't translate the requests into an intermediary format (such as ODBC).
- The client application connects directly to the database server. No translation or middleware layers are used, improving performance.

Disadvantages:

- Drivers are database dependent, as different database vendors use wildly different (and usually proprietary) network protocols.

Which Driver should be Used?

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

JDBC

SQL Revision

SQL

- Structured Query Language.
- Standardized syntax for “querying” (accessing) a relational database.
- It is database-independent.
- Actually, there are only some variations from DB to DB.

SQL Syntax

INSERT INTO *table* (*field1, field2*) VALUES (*value1, value2*)

- inserts a new record into the named table

UPDATE *table* SET (*field1* = *value1, field2* = *value2*) WHERE *condition*

- changes an existing record or records

DELETE FROM *table* WHERE *condition*

- removes all records that match the condition

SELECT *field1, field2* FROM *table* WHERE *condition*

- retrieves all records that match condition

Transactions

- Transaction = more than one statement which must all succeed (or all fail) together
- If one fails, the system must reverse all previous actions
- Also can't leave DB in inconsistent state halfway through a transaction
- COMMIT = complete transaction
- ROLLBACK = abort

JDBC

Creating Simple JDBC Applications

Some Interfaces & Classes

- **Statement:** The interface is used for representing the SQL statement. Example
  1. SELECT * FORM student_table;
  2. UPDATE student_table SET name = 'Neel' WHERE roll_no = '1';
  3. DELETE from students_table WHERE roll_no = '10';
- There are two specialised Statement type : **PrepredStatement** and **CallableStatement**.

5 Steps: To connect to a database

5 steps to connect any java application with the database in java using JDBC:

1. Register the driver class
2. Creating connection
3. Creating statement
4. Executing queries
5. Closing connection

5 Steps: Implementation

1. Register the driver class : The forName() method of Class class is used to register the driver class.

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

2. Creating connection : The getConnection() method of DriverManager class is used to establish connection with the database.

```
Connection con = DriverManager.getConnection
("jdbc:mysql://localhost:3306/ss",
 "root", "root");
```

3. Creating statement : 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();
```

5 Steps: Implementation (cont.)

4. Executing queries : 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));
}
```

5. Closing connection : 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();
```

Example 1: Connect to the MySQL DB

- We need to know following information 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/ss** 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.
  3. **Username:** The default username for the mysql database is **root**.
  4. **Password:** Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

JDBC

Statement interface

Statement interface

- Provides methods to execute queries with the database.
- It provides factory method to get the object of ResultSet.

Methods:

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.

Example 1:
First create a Table in the MYSQL Database & populate it with data

```
create database tc1;
use tc1;
create table emp
(
      id int(10),
      name varchar(40),
      age int(3)
);
```

Example 1.1: Code to Connect Java Application with mysql database

```
import java.sql.*;
class MysqlStatement1{
public static void main(String args[]){
try{
 Class.forName("com.mysql.jdbc.Driver");
 Connection con =
 DriverManager.getConnection( "jdbc:mysql://localhost:3306/tc1",
                                                "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);}
}
}
```

```
import java.sql.*;
class MysqlStatement1{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con =
 DriverManager.getConnection(
 "jdbc:mysql://localhost:3306/tc1",  "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);}
}  }
```

```
import java.sql.*;
class  MysqlStatement2{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con =
DriverManager.getConnection( "jdbc:mysql://localhost:33
06/tc1", "root", "root");
Statement stmt=con.createStatement();
int result = stmt.executeUpdate("delete from emp where
id=1");
System.out.println(result+" records affected");
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Example 1.2: Code to Connect Java Application with mysql database

```
import java.sql.*;
class  MysqlStatement2{
public static void main(String args[]){
try{
 Class.forName("com.mysql.jdbc.Driver");
 Connection con =
 DriverManager.getConnection( "jdbc:mysql://localhost:3306/tc1",
                                                    "root", "root");
 Statement stmt=con.createStatement();
 int result = stmt.executeUpdate("delete from emp where id=1");
 System.out.println(result+" records affected");
 con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Examples of JDBC URL (Connection String)

JDBC URL: The database are referenced by Java runtime engine using JDBC URL. The syntax of there URLs depends on the database driver you are using.

1. ODBC :

```
Syntax : jdbc:odbc:<data source name>
Example: jdbc:odbc:MyDataSource
```

2. MYSQL :

```
Syntax : jdbc:mysql://[host][:port]/[database]
Example: jdbc:mysql://localhost:3306/Mydatabase
```

3. ORACLE:

```
Syntax :
jdbc:oracle:<drivertype>:<user>/<password>@<database>
Example:
jdbc:oracle:thin:myuser/mypassword@localhost:mydatabase
```

Example 1.3: Creating a Table in SQL using Java

In Program 1 change the SQL statement

//4. Execute query:

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

TO

//4. Execute query:

```
int i = stmt. executeUpdate
   ( "create TABLE Students(Roll no int,
     Name Varchar(20))" );
if(i==0)
 System.out.println("Table Not created");
else
 System.out.println("Table created");
```

Example 1.4: Updating data in database using Java

//3.Create the statement object which is
//used to execute query in database
```
   Statement st = con.createStatement();
```

//4. Execute query
```
int i = st.executeUpdate("UPDATE emp SET 
name ='AMIT' WHERE id = 1" );
if(i==0)
 System.out.println("Table Not updated");
else
 System.out.println("Table updated");
```

JDBC

PreparedStatement

Prepared Statement 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(?,?,?)";
```

- Here we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

- **Improves performance**: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

Prepared Statement instance:

- 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 qry)
throws SQLException { }
```

Methods of Prepared Statement interface:

Method	Description
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.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.

Example 2: Prepared Statement that inserts a record

```
import java.sql.*;
import java.util.Scanner;
public class PrepareStmt
{
public static void main(String[] args) {
try {
 //1. Registration
 Class.forName("com.mysql.jdbc.Driver");
 //2. Connection object
 Connection con =
DriverManager.getConnection("jdbc:"+
"mysql://localhost:3306/ss","root", "root");
 //3. Statment object to execute query
 PreparedStatement st =
 con.prepareStatement(
 "insert into emp values(?,?,?)");
```

Example 2: Prepared Statement that inserts a record (Cont.)

```
//4. execute query
 st.setInt(1,3);
 st.setString(2,"ravi");
 st.setInt(3,12);
 int val = st.executeUpdate();
 System.out.println("Records inserted: "+val);
 //5. close connection
 st.close();
}
catch (Exception e)
{    System.out.println(e);        }
} // Main ends
} // Class ends
```

```
import java.sql.*;
class MysqlStatement1{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con =
 DriverManager.getConnection(
 "jdbc:mysql://localhost:3306/tc1",  "root", "root");
Statement stmt=con.createStatement();
int result =
stmt.executeUpdate(" insert into emp values (3, ‘Ravi’,
12)");
System.out.println("Records inserted: "+result);
con.close();
}catch(Exception e){ System.out.println(e);}
}  }
```

```
import java.sql.*;
class  MysqlStatement1{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con =
DriverManager.getConnection( "jdbc:mysql://localhost:33
06/tc1", "root", "root");
PreparedStatement st =  con.prepareStatement(
 "insert into emp values(?,?,?)");
st.setInt(1,3);  st.setString(2,"ravi");   st.setInt(3,12);
 int val = st.executeUpdate();
 System.out.println("Records inserted: "+val);
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Example 3: Prepared Statement that updates a record

```
PreparedStatement stmt =
con.prepareStatement("update emp set name=?
where id=?");

stmt.setString(1,"AJP");
stmt.setInt(2, 1);

int i = stmt.executeUpdate();
System.out.println(i+" records updated");
```

Example 4: Problem Statement

Problem Statement

- Consider bank table with attribute AccountNo, CustomerName, Balance, Phone and Address.
- Write a database application which allows insertion, updation and deletion of records in Bank Table. Print values of all customer's.

Example 4: Code

```
import java.sql.*;
import java.util.Scanner;
public class DemoJdbc {
public static void main(String[] args) {
try {
//1. Register the driver
Class.forName("com.mysql.jdbc.Driver");


//2. establish connection by con object
Connection con = 
DriverManager.getConnection("jdbc:" + 
"mysql://localhost:3306/ss","root", "root");
System.out.println("Connection established");
```

Example 4: Code (cont.)

```
//3. Create the statement object which is used
to execute query in database
Statement st = con.createStatement();

//4. Execute query
int ch;
Scanner s = new Scanner(System.in);
System.out.println("1. To insert data");
System.out.println("2. To update data");
System.out.println('"3. To delete data");
System.out.println("otherwise : To fetch all
the data in the table");
System.out.println("Enter your choise :");
ch = s.nextInt();
```

Example 4: Code (cont.)

```
switch(ch){
// To insert data
case 1:
{
 int c1 = st.executeUpdate (
 "INSERT into bank_tb Values
 (10,'mukesh',25000,876789876,'Rajkot') " );
 System.out.println("Data inserted.");
 break;
}
```

Example 4: Code (cont.)

```
// To update data
case 2:{
 int i = st.executeUpdate("UPDATE bank_tb SET
 customerName ='AMIT' WHERE accountNo = 1 ");
 if(i==0)
   System.out.println("Table Not updated");
 else
   System.out.println("Table updated");
 break;
}
// To delete data
case 3:{
 st.executeUpdate("DELETE FROM bank_tb
 WHERE accountNo = 1");
 break;
}
```

Example 4: Code (cont.)

```
default :
{
 ResultSet rs1 = st.executeQuery(
 "SELECT * FROM bank_tb");
 while(rs1.next())
 System.out.println(rs1.getInt(1)
 + "   " + rs1.getString(2)+ "   "
 + rs1.getInt(3) + " “ + rs1.getInt(4)
 + " " + rs1.getString(5));
 break;
}
} // End of Swtich
```

Example 4: Code (cont.)

```
//5.close the connection
con.close();
} // End of try clause
catch (Exception e)
{
     System.out.println("Exception: " + e);
}
} // End of main
} // End of class
```

JDBC

CallableStatement

Java Callable Statement Interface

- CallableStatement interface is used to call
  - **The stored procedures and functions.**
- 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.
- To call the stored procedure, you need to create it in the database.

```
create procedure "INSERTR"
     (id IN NUMBER,   name IN VARCHAR2)
is
begin
     insert into user1 values(id,name);
end;
```

Example 5: Java Callable Statement Interface

- To call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user1.

```
import java.sql.*;
public class CallProc {
public static void main(String[] args)
throws Exception
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection(
     "jdbc:oracle:thin:@localhost:1521:xe",
     "root", "root");
CallableStatement stmt =
     con.prepareCall( "{call insertR(?,?)}" );
stmt.setInt(1,101);
stmt.setString(2,"Amit");
stmt.execute();
System.out.println("success");
}
}
```

JDBC

Exploring ResultSet Operations

ResultSet interface

- JDBC provides the following connection methods to create statements with desired ResultSet –
  - **createStatement(int RSType, int RSConcurrency);**
  - **prepareStatement(String SQL, int RSType, int RSConcurrency);**
  - **prepareCall(String sql, int RSType, int RSConcurrency);**
- The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

- The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	**Description**
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

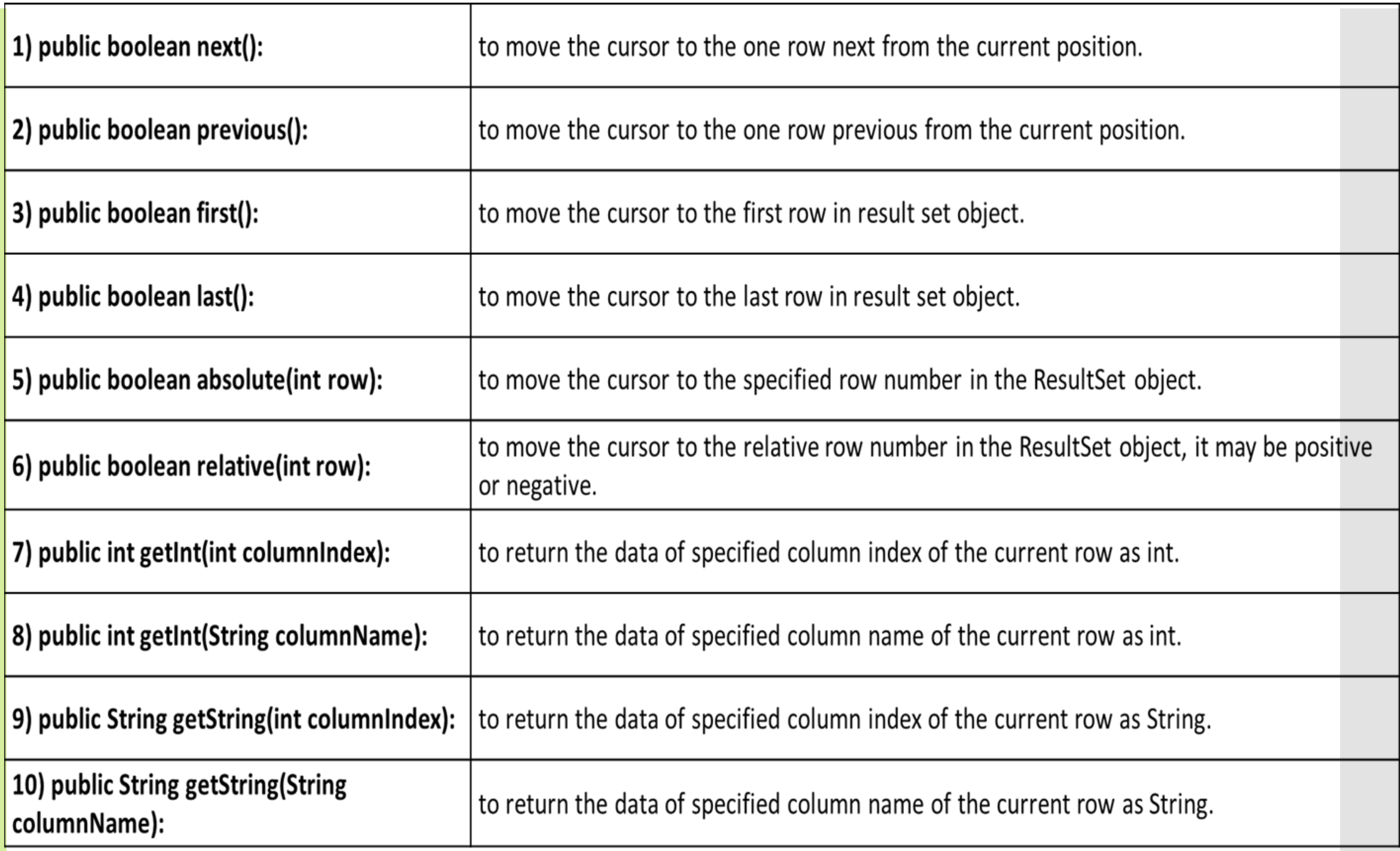
Concurrency of ResultSet

- The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	**Description**
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

- All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object –
  - Statement stmt = conn.createStatement (ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);

Methods of ResultSet interface



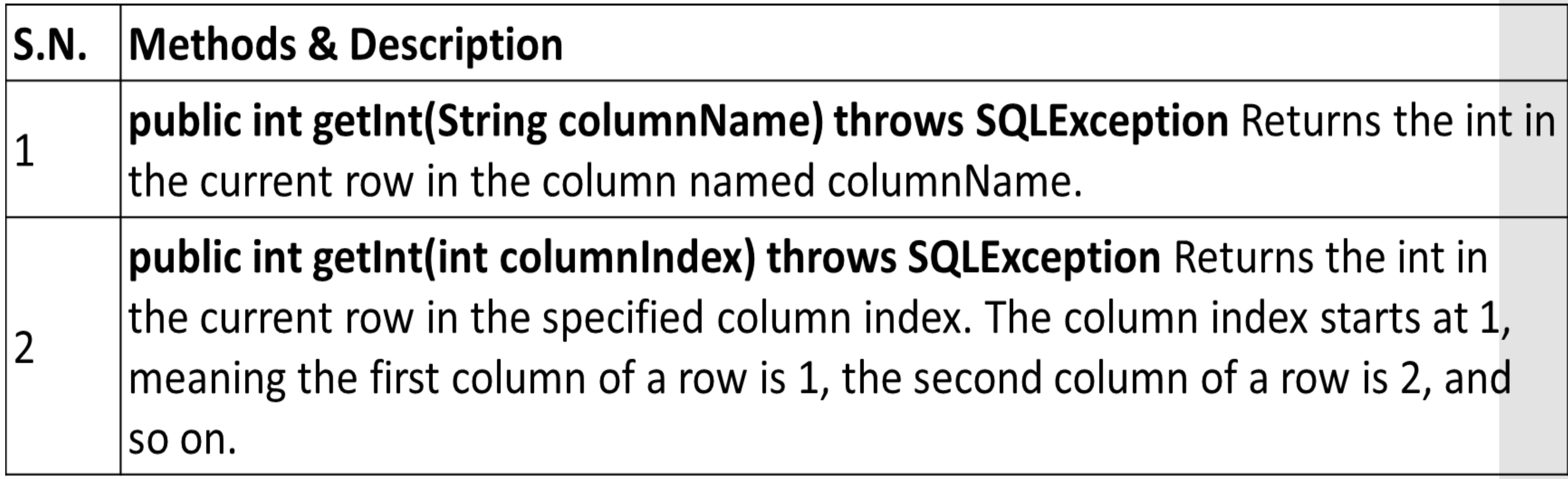
1) public boolean next():	to move the cursor to the one row next from the current position.
2) public boolean previous():	to move the cursor to the one row previous from the current position.
3) public boolean first():	to move the cursor to the first row in result set object.
4) public boolean last():	to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	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):	to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	to return the data of specified column name of the current row as String.

Viewing a Result Set

- The ResultSet interface contains dozens of methods for getting the data of the current row.
- There is a get method for each of the possible data types, and each get method has two versions –
    - One that takes in a column name.
    - One that takes in a column index.
- For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –
- Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

Viewing a Result Set

- There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob.



S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a Result Set

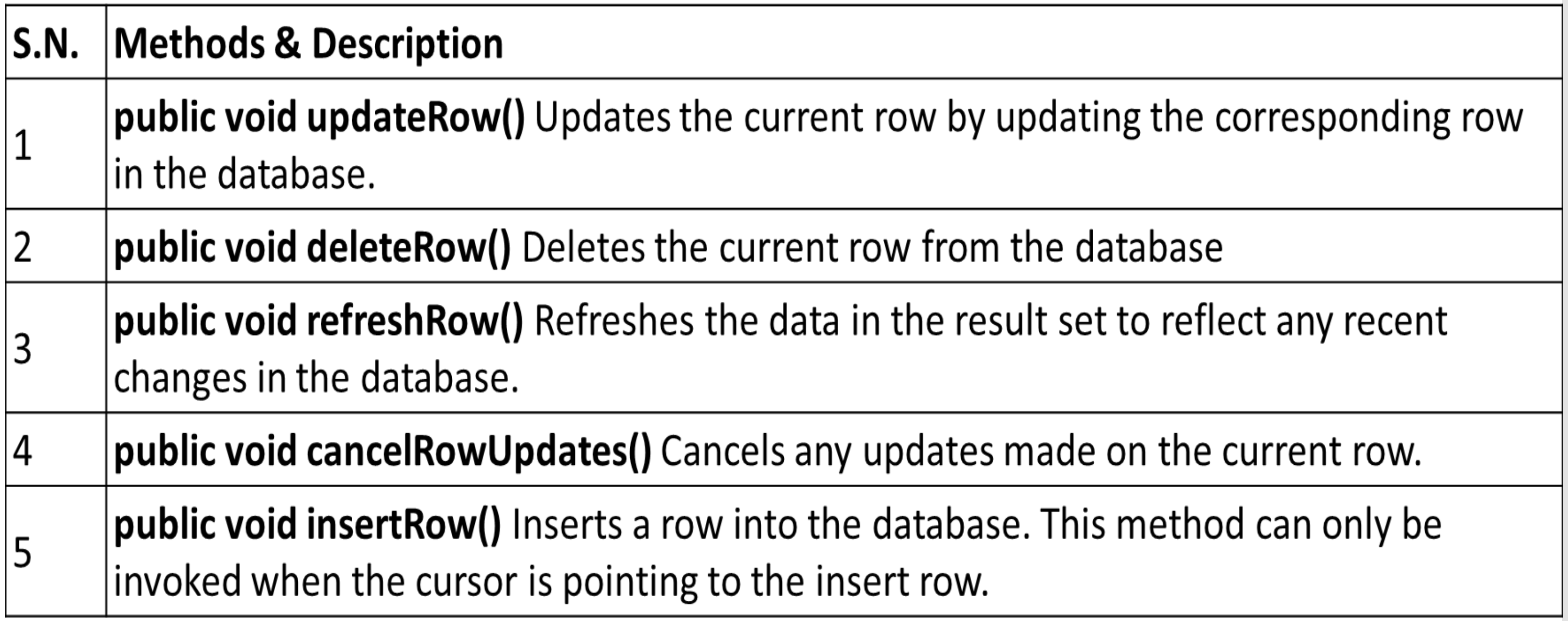
- The ResultSet interface contains a collection of update methods for updating the data of a result set.
- As with the get methods, there are two update methods for each data type –
  - One that takes in a column name.
  - One that takes in a column index.
- There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.
- For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods

Updating a Result Set

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

Updating a Result Set

- Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.



S.N.	Methods & Description
1	**public void updateRow()** Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()** Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

Example 6: Scrollable ResultSet

```
import java.sql.*;
public class ScrollTo3rdRecord {
public static void main(String args[])throws E
xception{
Class.forName("com.mysql.jdbc.Driver");
Connection con =  DriverManager.getConnection
("jdbc:mysql://localhost:3306/ss","root","");
Statement stmt = con.createStatement
     (ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
ResultSet rs =
stmt.executeQuery("select * from emp");
//getting the record of 3rd row
rs.absolute(3);
System.out.println(rs.getString(1) + " " +
rs.getString(2) + " " + rs.getString(3));
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.

Example 7: **ResultSetMetaData**

```
import java.sql.*;
class Rsmd{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mu", "root","");
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.getColumnTypeN
ame(1));
con.close();
}catch(Exception e){ System.out.println(e);}  }
}
```

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.

Example 8: Access data from Oracle DB

```
import java.sql.*;
public class OracleDBExample {
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 of 1st column: "+rsmd.getColumnTypeName(1));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Example 8: Output

```
Total columns: 2
Column Name of 1st column: ID
Column Type Name of 1st column: NUMBER
```

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.

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.

Example 9: **DatabaseMetaData**

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

JDBC

Batch updates in JDBC

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.
- The required methods for batch processing are given below:

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

Example of batch processing in JDBC

- Simple examples of batch processing:
- It follows following steps:
  - Load the driver class
  - Create Connection
  - Create Statement
  - Add query in the batch
  - Execute Batch
  - Close Connection

Example 10: Batch processing in JDBC

```
import java.sql.*;
class BatchUpdateEx{
public static void main(String args[])throws Exception{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection("jdbc:mys
ql://localhost:3306/mu", "root", "");
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("insert into emp values(10,'abhi',25)");
stmt.addBatch("insert into emp values(11,'Umesh',35)");
stmt.executeBatch(); //executing the batch
con.commit();
con.close();
}
}
```

JDBC

Using RowSet Objects

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

Advantages:

- It is easy and flexible to use
- It is Scrollable and Updatable bydefault

How to create and execute RowSet?

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

rowSet.setUrl("jdbc:oracle:thin:@localhost:152
1:xe");

rowSet.setUsername("system");

rowSet.setPassword("oracle");

rowSet.setCommand("select * from emp");

rowSet.execute();
```

Example 11: JdbcRowSet

```
import java.sql.*;
import javax.sql.rowset.*;
public class RowSetExample
{
public static void main(String[] args)
throws Exception
{
 Class.forName("com.mysql.jdbc.Driver");
 //Creating and Executing RowSet
 JdbcRowSet rowSet = RowSetProvider.newFactory
().createJdbcRowSet();
 rowSet.setUrl("jdbc:mysql://localhost:3306/mu
");
 rowSet.setUsername("root");
 rowSet.setPassword("");
```

Example 11: JdbcRowSet (cont.)

```
  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));
  }
 }
}
```

JDBC

Managing Database Transactions

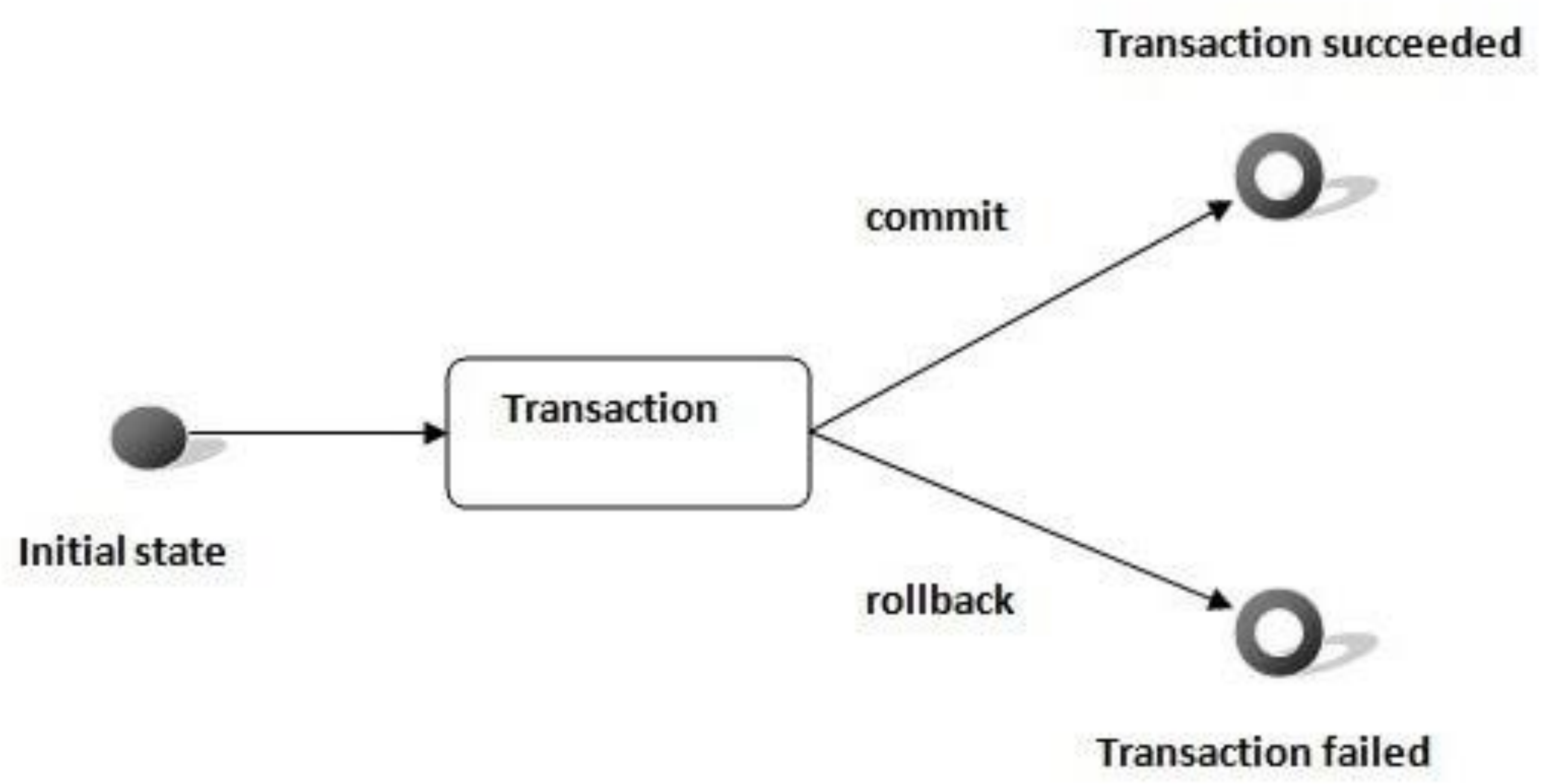
Transaction Management in JDBC

- Transaction represents **a single unit of work**.
- The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.
  1. **Atomicity** means either all successful or none.
  2. **Consistency** ensures bringing the database from one consistent state to another consistent state.
  3. **Isolation** ensures that transaction is isolated from other transaction.
  4. **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 is **fast performance** It makes the performance fast because database is hit at the time of commit.

Methods to manage transaction

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.

Transaction State diagram



Example 12: Manage transaction

```
import java.sql.*;
public class DemoJdbc {
public static void main(String[] args) {
try {
       Class.forName("com.mysql.jdbc.Driver");
       Connection con = DriverManager.getConnection("jdbc:"
       + "mysql://localhost:3306/ss", "root", "root");
       System.out.println("Connection established");
       con.setAutoCommit(false);
       Statement stmt=con.createStatement();
       stmt.addBatch("insert into emp values(10,'abhi',25)");
       stmt.addBatch("insert into emp values(11,'Umesh',35)");
       stmt.executeBatch(); //executing the batch
       con.commit();  // OR con.rollback();
       con.close();
     } catch (Exception e)
     {  System.out.println("Exception: " + e);     }
}
}
```