Unit 8
JDBC

Introduction

- The success of any business depends on the information and data assets it possesses.
- Therefore it is very important for a business to have a system for storing and retrieving data.
- Though files are used for storing data, they are highly inefficient in many respects like the amount of data that can be stored, retrieving the data and many more.
- In order for the businesses to be competent, they need sophisticated systems for storing complex and heavy volume of data and at the same time retrieve the data faster enough to process the data. One such systems are the so called Databases.

Database Basics

- Databases store the data in the form of tables with each table containing several records of data.
- Databases are broadly categorized into two types as listed below:
  - Single table databases and
  - Multi table or relational databases.
- A single table databases doesn’t mean that they have just one table. It will have several tables of data and all the tables are independent of each other.
- The main drawback with these databases is the lack of flexibility while retrieving the data.
- On the other hand in multi table databases, two or more tables are related with other. This feature offers more flexibility and power to retrieve the data.
- These databases are therefore called as relational databases. Most of the modern enterprises use

- Following figure shows how the data is stored in databases:

Name	**SSN**	**Age**	**Country**
John	1234567	47	USA
Smith	343534 5	21	Canada

Table:Customer

- The above table shows customers table that has two records of data. A typical database will have ‘n’ number tables with each having ‘n’ number of records.

- A relational database can store the data in several tables that are related with each other as shown below:

Name	SSN	Age	Country
John	12345	47	USA
Smith	34353	21	Canada

Table:Customer

SSN	AddressLine 1	AddressLine 2	City	State
1234567	Apt # 20	1111 S St	LA	CA
3435345	Apt #2345	2222 N St	Detroit	MI

Table:Address

- If you notice the above tables, they are related with each other by the common column SSN.
- The two tables together represent the customer information. To pull the customer information, we need to pull the data from both the tables based on the matching SSN.
- Now that we know what relational databases are capable of, we need to know something about how to store the data and retrieve the data to and from databases.
- This is where SQL comes into picture. So, let’s see what this is all about.

Structured Query Language (SQL)

- Structured Query Language which is abbreviated as SQL is a standard query language for working with relational databases.
- SQL is used for creating the tables, populating the tables with data, manipulating the data in the tables, deleting the data in the tables and the tables itself.

Creating a table

- Let’s create a table named Customers to store customer data like firstname, lastname, ssn, age, city, state, and country as shown below:

FirstName	LastName	SSN	Age	City	State	Country

The SQL syntax for creating a table is shown below:

```
CREATE TABLE <TABLE NAME>
(
COLUMN 1 NAME    COLUMN 1 TYPE,
COLUMN 2 NAME    COLUMN 2 TYPE,
 .
 .
COLUMN N NAME    COLUMN N TYPE
);
```

- Using the above syntax, to create the customers table, the SQL query will be as shown below:

```
CREATE TABLE Customers
(
FirstName       VARCHAR(50),
LastName        VARCHAR (50),
SSN             NUMERIC(10),
Age             NUMERIC(10),
City            VARCHAR (32),
State           VARCHAR (2),
Country         VARCHAR (32)
);
```

- While creating table, it’s very important to mention the type of data that will be stored in various columns.
- VARCHAR is a standard datatype for storing text data.
- NUMERIC is used for storing numbers. The numbers in parenthesis represent the maximum length of data that can be stored.
- Likewise, there are several other data types.

SQL Data Type	**Description**
VARCHAR	Used for storing text data
NUMERIC	Used for storing numbers
DATE	Used for storing dates

Inserting the data into the table

- To insert the data into the tables, we use the INSERT query.
- A single insert query inserts one record or row of data. Following is the sql syntax.
- INSERT INTO <TABLE NAME> VALUES (COL1 DATA, COL2 DATA …. COL (N) DATA);
- Using the above syntax, we insert the record into customers table as

  INSERT INTO Customers

  VALUES

  (‘Smith’, ‘John’, 123456, 20, ‘Vegas’, ‘CA’, ‘USA’);
- The above SQL inserts the data in the Customers table as shown below:

FirstName	LastName	SSN	Age	City	State	Country
Smith	John	123456	20	Vegas	CA	USA

While using the INSERT sql,

- Every data must be separated by a comma,
- All the text data must be enclosed in single quotes,
- Numeric data must not be enclosed in single quotes.

If you don’t know what value to insert in a particular column, you must specify a NULL as shown below:

```
INSERT INTO Customers
VALUES
(‘Smith’, null, 123456, 20, null, ‘CA’, ‘USA’);
```

Updating records in table

- To update the data in an existing record, we use the UPDATE query statement. The syntax for this query is shown below:

  UPDATE <TABLE NAME> SET

  COLUMN 1 NAME = NEW COLUMN 1 VALUE,

  COLUMN 2 NAME = NEW COLUMN 2 VALUE

  ...

  WHERE

  COLUMN 1 NAME = VALUE 1 AND

  COLUMN 2 NAME = VALUE 2 ……..

- Let’s say we want to change the last name to ‘Roberts’ and first name to ‘Joe’ whose SSN is 123456. The update query will be,

  UPDATE CUSTOMERS SET

  FirstName = ‘Joe’, LastName = ‘Roberts’

  WHERE SSN = 123456

Retrieving records from table

- Retrieving the records is the most important database operation. We use the SELECT query to retrieve the records. The syntax for this query is as shown below:
- SELECT COL 1 NAME, COL2 NAME …… FROM <TABLE NAME> WHERE <CONDITION>
- **Case 1: Retrieve all the customers data**

  SELECT * from Customers;
- In the above sql asterisk(*) means all columns of data. Since there is no WHERE clause in the query, it will pull up all the records. The result of the above query is the entire table.

- **Case 2: Select all the customers who belong to the state of CA.**

SELECT * FROM Customers WHERE State = ‘CA’;

- **Case 3: Retrieve the first names and last names whose country is USA and state is CA (multiple conditions)**

SELECT firstName, lastName from Customers WHERE State=’CA’ AND Country=’USA’;

- **Case 4: Let's say we have the following two tables.**

Publisher	**ISBN**	**Title**
Wrox	AB3456	Perl Scripting
McGraw	YZ6789	Core Java

Table: Books

Author	**ISBN**	**Country**
Smith	AB3456	Germany
John	YZ6789	Australia

Table: Authors

- If you noticed the above two (Books and Authors) tables, they are joined together using the ISBN column.
- If we need to pull the publisher, author, title and country (data in both the tables) for a particular ISBN, we do so as shown below:

  SELECT t1.Publisher, t1.Title, t2.Author, t2.Country from Books t1, Authors t2 WHERE t1.ISBN = t2.ISBN;
- Whenever we need to pull data from multiple tables, we need to use alias names for tables.
- In the above query, t1 is the alias name for Books table, and t2 is the alias name for Authors table. Therefore, all the columns in books table must be prefixed with “t1.” and columns in authors table must be prefixed with “t2.” as shown below:

  SELECT t1.Publisher, t1.Title, t2.Author, t2.Country

- Now, look at the where clause. Since we need to pull

Deleting the records

- To delete the records in the table we use DELETE query. The syntax for this query is shown below:

  DELETE FROM <TABLE NAME> WHERE <CONDITION>

- For instance, to delete all the customers whose state is CA, the query will be,

  DELETE FROM Customers where State=’CA’;

- The above SQL deletes 3 records in the customers table.
- These are all the basic SQL operations you need to know to work with any database.

JDBC

- JDBC stands for Java Database Connectivity which is a technology that allows Java applications to,

1. Interact with relational databases
2. Execute SQL queries against the databases

- To better understand JDBC technology, look at the following figure:

To better understand JDBC technology, look at the following figure:

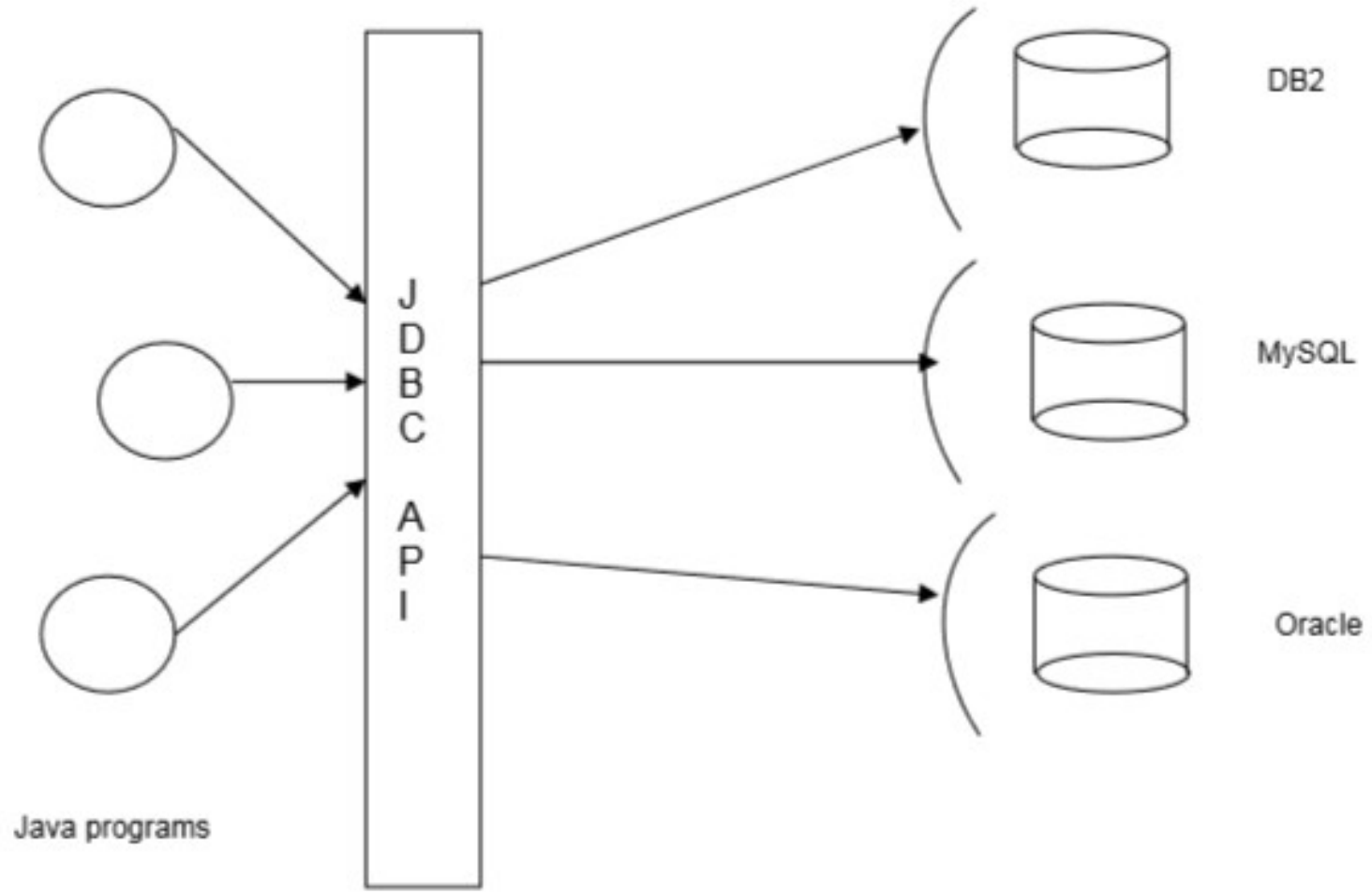


Fig 11.1 JDBC and Databases

How JDBC talks with Databases

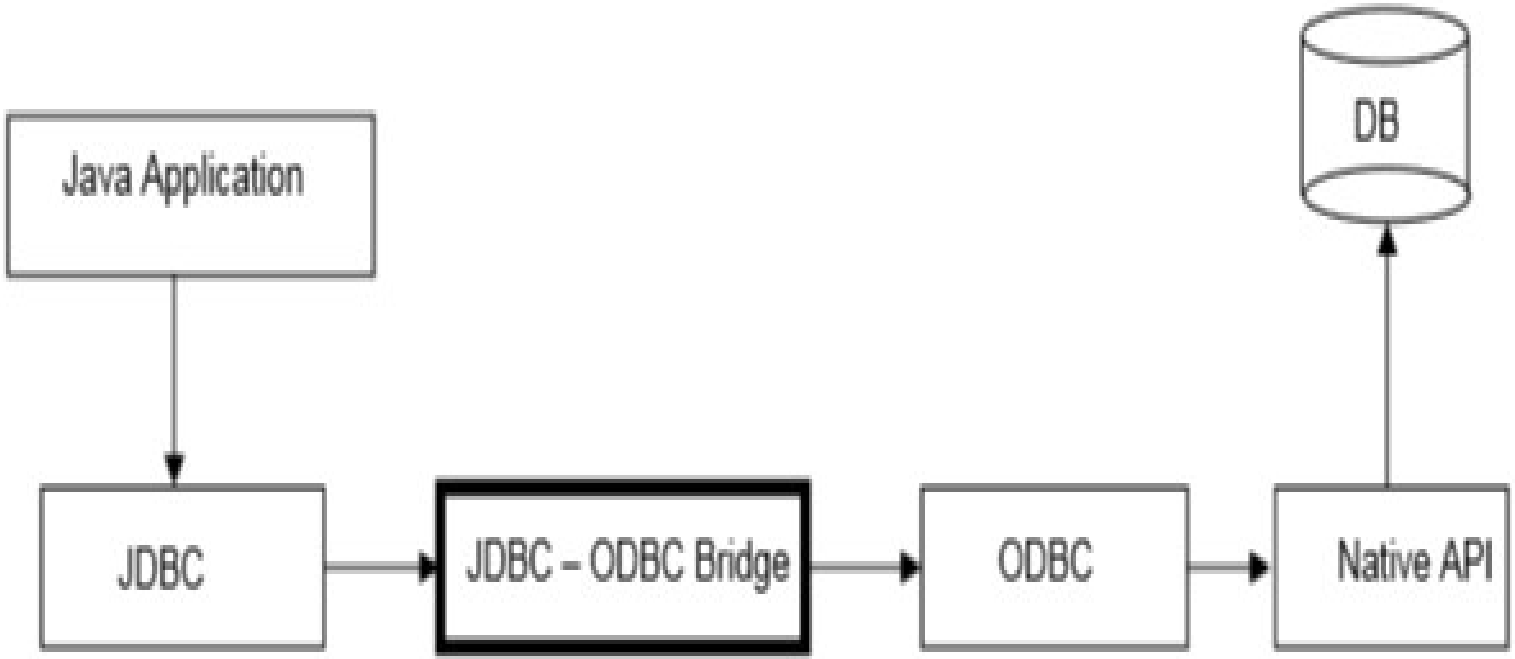
- Databases are proprietary applications that are created by noted companies like Oracle, Microsoft and many more.
- Every database will expose something called Driver through with one can interact with database for SQL operations.
- A driver is like a gateway to the database.
- Therefore, for any Java application to work with databases, it must use the driver of that database.

Database Drivers

- A driver is not a hardware device. It’s a software program that could be written in any language.
- Every database will have its own driver program.
- Given a driver program of a particular database, the challenge is how to use it to talk with database. To understand this, we need to know the different types of drivers.
- There are basically 4 different types of database drivers as described below:

1. JDBC-ODBC Bridge Driver

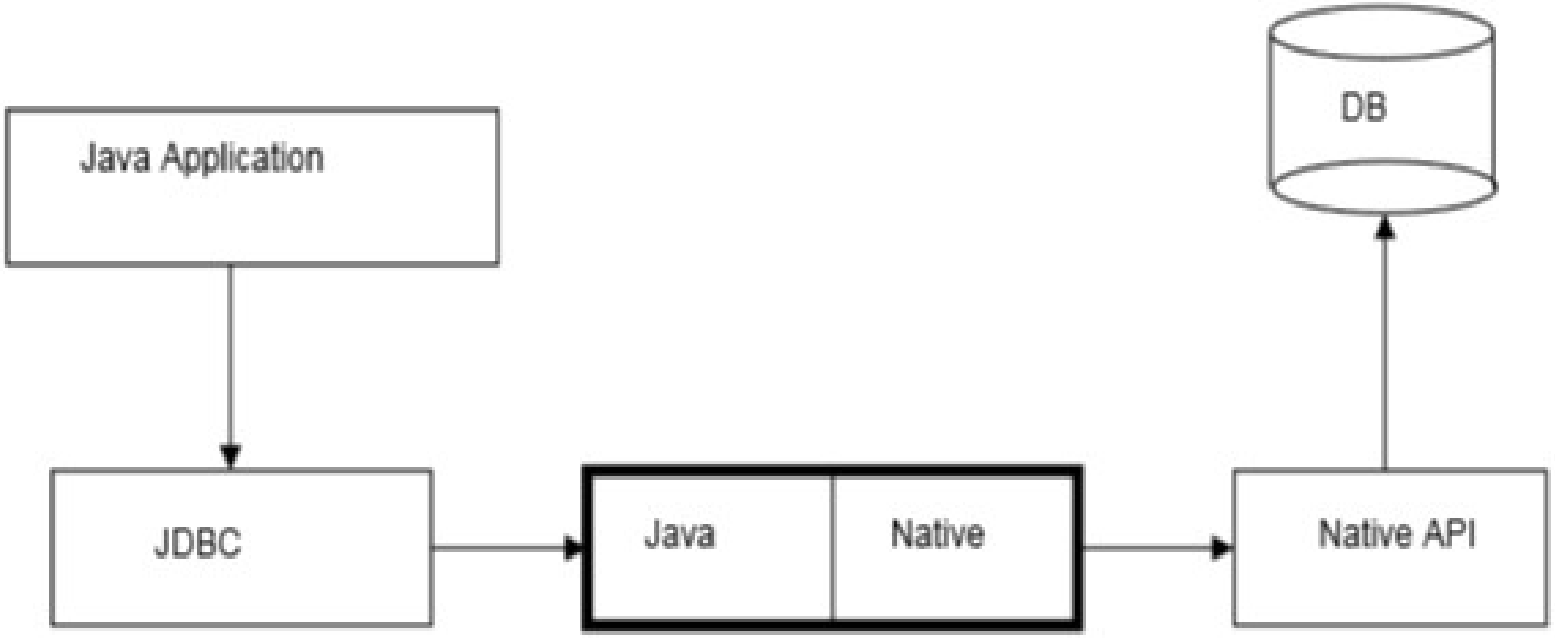
- This is also called as Type-1 driver. Fig 11.2 shows the configuration of this driver.



- As you can notice from the above figure, this driver translates the JDBC calls to ODBC calls in the ODBC layer.
- The ODBC calls are then translated to the native database API calls.
- Because of the multiple layers of indirection, the performance of the application using this driver suffers seriously.
- This type of driver is normally used for training purposes and never used in commercial applications.
- The only good thing with this driver is that you can access any database with it.
- In simple words, with this driver for JDBC to talk with vendor specific native API, following translations will be made.
- JDBC - > ODBC -> Native

2. Partly Java and Partly Native Driver

- This is also called as Type-2 driver. As the name suggests, this driver is partly built using Java and partly with vendor specific API. Figure below shows the configuration of this driver



- With this driver, the JDBC calls are translated to vendor specific native calls in just one step.
- It completely eliminates the ODBC layer.
- Because of this ODBC layer elimination, the applications using this driver perform better.

3. Intermediate Database access Driver Server

- This is also known as Type-3 driver.
- If you look at the Type-2 driver configuration, we only access one database at a time.
- However there will be situations where multiple Java programs need to access to multiple databases. This is where Type-3 driver is used.
- It acts as a middleware for the applications to access several databases.
- The Type-3 configuration internally may use Type-2 configuration to connect to database.

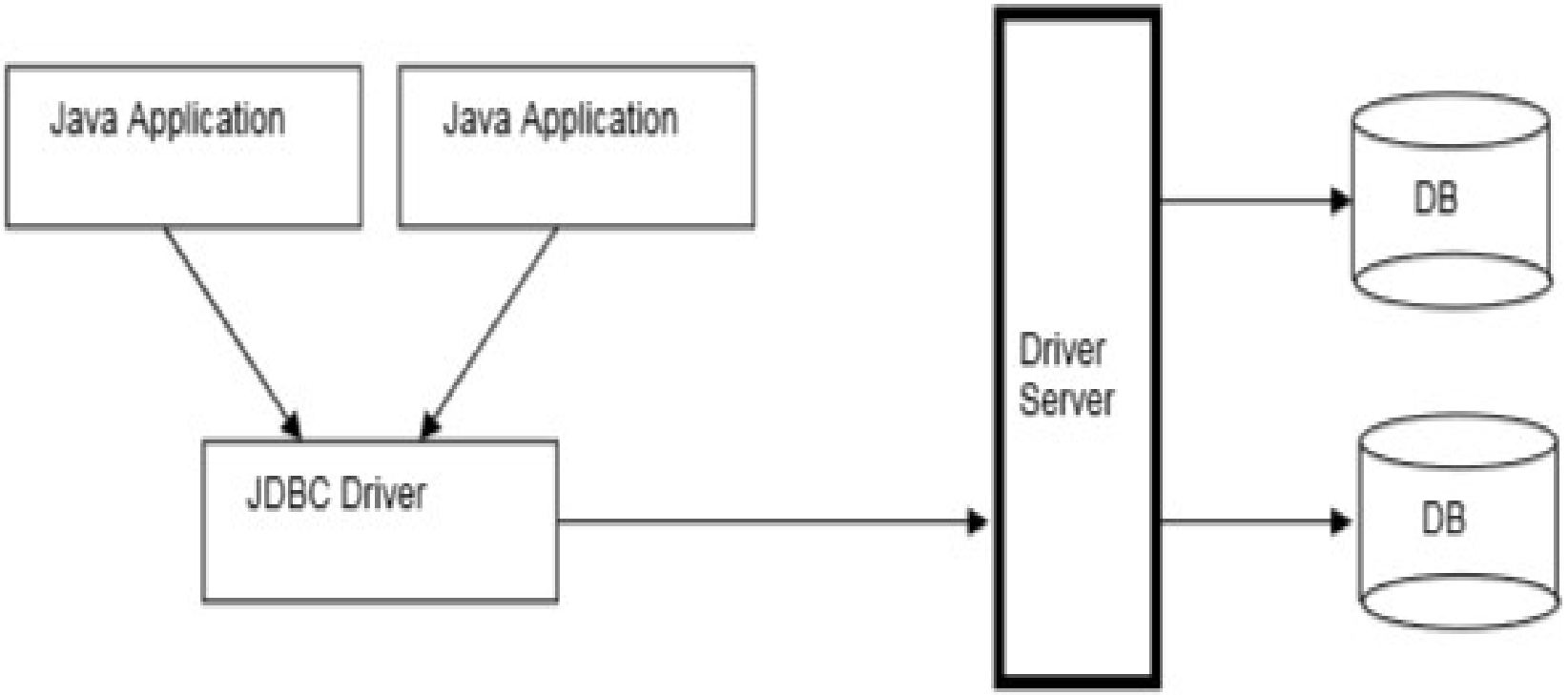
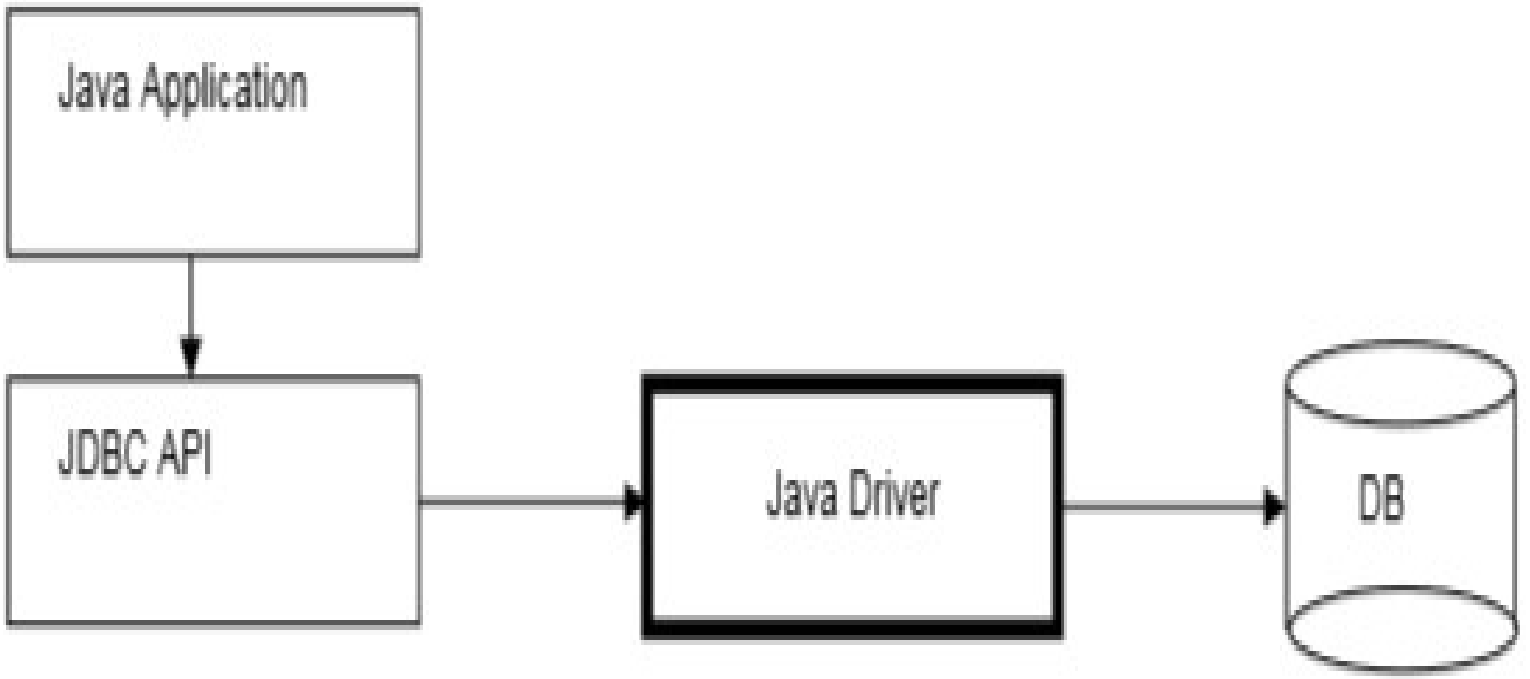


Fig 11.4 Driver Server

4. Pure Java Drivers

- These are called as Type-4 drivers.
- Because of the widespread usage of Java, to facilitate easy and faster access to databases from Java applications, database vendors built the driver itself using Java like good friends helping each other.
- This is really cool as it completely eliminates the translation process.
- JDBC is Java based technology, and with the driver also written in Java, it can directly invoke the driver program as if it were any other Java program.
- This is the driver that all the J2EE applications use to interact with databases.
- Because this is a Java to Java interaction, this


Java Application
JDBC API
Java Driver
DB

JDBC API

- JDBC technology is nothing but an API.It is a set of classes and interfaces that our Java programs use to execute the SQL queries against the database.
- The fundamental idea behind JDBC is that, Java programs use JDBC API, and JDBC API will in turn use the driver to work with the database.
- Therefore, to work with MySQL database, we need to configure the JDBC with all the MySQL information. This information is nothing but:
  1. Name of the MySQL driver class
  2. URL of the database schema.
- In simple words, for a Java program to work with the database, we first need to configure JDBC with the above database info and then make the Java program use the JDBC.
- Table lists the important JDBC classes and interfaces

Class/Interface	Description
`DriverManager`	This classes is used for managing the Drivers
`Connection`	It's an interface that represents the connection to the database
`Statement`	Used to execute static SQL statements
`PreparedStatement`	Used to execute dynamic SQL statements
`CallableStatement`	Used to execute the database stored procedures
`ResultSet`	Interface that represents the database results
`ResultSetMetaData`	Used to know the information about a table
`DatabaseMetadata`	Used to know the information about the database
`SQLException`	The checked exception that all the database classes will throw.

- JDBC programming is the simplest of all. There is a standard process we follow to execute the SQL queries. Following lists the basic steps involved in any JDBC program.
  1. Load the Driver
  2. Get the connection to the database using the URL
  3. Create the statements
  4. Execute the statements
  5. Process the results
  6. Close the statements
  7. Close the connection.

```
import java.sql.Connection;
import java.sql.DriverManager;

public class ConnectionTest {
    public static void main(String args[]) {
        try {
// Step 1: Load the Driver
            Class.forName("com.mysql.jdbc.Driver");
// Step 2: Get the connection by passing the URL
            String url = "jdbc:mysql://localhost:3306/MyDB";
            Connection con =
DriverManager.getConnection(url,"root","");
            System.out.println(con);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Creating Table

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class CreatingTable {
    public static void main(String args[]) throws Exception
{
        createTable();
    }
    private static void createTable() throws Exception {
        String sql = "CREATE TABLE CUSTOMERS(" +
                "FIRSTNAME VARCHAR(50),MIDDLENAME
VARCHAR(20),"
                + "LASTNAME VARCHAR(50),AGE NUMERIC(2),"
                + "SSN NUMERIC(10),CITY VARCHAR(32)"
                + ",STATE VARCHAR(20),COUNTRY
```

```
// Get the connection using our utils.
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
        Connection con = DriverManager.getConnection(url,
"root", "");
        if (con != null) {
// Create statement from connection
            Statement stmt = con.createStatement();
// Execute the statement by passing the sql
            int result = stmt.executeUpdate(sql);
            if (result != -1) {
                System.out.println("Table created
sucessfully");
            } else {
                System.out.println("Could'nt create table.
Please check your SQL syntax");
            }
```

```
// Close the statements and Connections
            stmt.close();
            con.close();
        } else {
            System.out.println("Unable to get the
connection");
        }
    }
}
```

First Name	Middle Name	Last Name	Age	SSN	City	State	Country
Ram	K	Shrestha	34	112233	Kathmandu	3	Nepal

Insert Data

```
import java.sql.*;
import javax.swing.JOptionPane;

public class Insert {
    Connection con;
    Statement stmt;
    public void insertData() {
        try {
            String sql = "INSERT INTO CUSTOMERS
VALUES('Ram','K','Shrestha',34,112233,'kathmandu'
,'3','Nepal')";
            Class.forName("com.mysql.jdbc.Driver");
            String url =
"jdbc:mysql://localhost:3306/MyDB";
```

```
stmt = con.createStatement();
            int result = stmt.executeUpdate(sql);
            if (result != -1) {
                JOptionPane.showMessageDialog(null,
"Insertion Successful");
            } else {
                JOptionPane.showMessageDialog(null,
"Error in Insertion");
            }
            stmt.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
 public static void main(String[] args) {
        Insert i = new Insert();
        i.insertData();
    }

}
```

Inserting Data

```
import java.sql.*;


public class InsertingData {
    public static void main(String args[]) throws
Exception {
        String sql = "INSERT INTO CUSTOMERS
VALUES(" + "'" +                    firstName + "' ,"+ "'"
+ middleName + "'," + "'"
                        + lastName + "' ,"+ age + ","
                        + ssn + " ," + "'"+ city + "' ," +
"'"
                        + state + "'," + "'"+ country +
"')";
// Pass the sql to JDBC method
```

```
private static void insertCustomer(String sql) throws
Exception {
// Get the connection using our utils.;
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
        Connection con = DriverManager.getConnection(url,
"root", "");
        if (con != null) {
// Create statement from connection
            Statement stmt = con.createStatement();
// Execute the statement by passing the sql
            int result = stmt.executeUpdate(sql);
            if (result != -1) {
                System.out.println("Inserted" + result +
"Record(s) successfully");
            } else {
                System.out.println("Unable to insert record. Please
check                                   your SQL syntax");
            }
```

```
// Close the statements and Connections
            stmt.close();

            con.close();
        } else {
            System.out.println("Unable to get the
connection");
        }
    }
}
```

```
import java.sql.*;
public class ReadingData {
    static Connection con = null;
    static Statement stmt = null;
    static ResultSet rs = null;
    public static void main(String args[]) throws Exception
{
        readData();
    }
    private static void readData() throws Exception {
        // Create the SQL
        String sql = "SELECT * FROM CUSTOMERS";
        // Get the connection using our utils.
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
        Connection con = DriverManager.getConnection(url,
"root", "");
```

```
if (con != null) {
// Create statement from connection
            stmt = con.createStatement();
// Execute the statement by passing the sql.
            rs = stmt.executeQuery(sql);
// Iterate over all the returned records
            while (rs.next()) {
                String firstName = rs.getString(1);
                String middleName = rs.getString(2);
                String lastName = rs.getString(3);
                int age = rs.getInt(4);
                int ssn = rs.getInt(5);
                String city = rs.getString(6);
                String state = rs.getString(7);
                String country = rs.getString(8);
```

```
System.out.println(firstName + " ," + middleName + " ," 
+ lastName + " ," + age + " ," + ssn + " ," + city + " ," 
+ state + " ," + country);
            }// End of While
        } else {
            System.out.println("Unable to get the 
connection");
        }
    }
}
```

- As you can see from the above code, we used the executeQuery() method to execute the SELECT query.
- This method will return a ResultSet object as shown below:

- To read the records, we need to move the pointer to the record using the next() method until the pointer reaches the end of the records.
- This is what the while loop does.
- Once we are in the loop, we need to read the record that the pointer points using the get methods.
- The get methods can either take the column number starting from 1 or the column name itself.
- In our example we used the column numbers. Following are the most commonly used get methods:
  - public String getString() Used for VARCHAR columns
  - public int getInt() Used for NUMERIC columns
  - public Date getDate() Used for DATE columns.
- Based on the above methods, since SSN and Age are declared as NUMERIC data types when we first created the table, we used getInt() method to read the data, and all the remaining are VARCHAR types, so we used getString() method to read the column data as shown

- Instead of specifying the column numbers, we can also specify the column names as shown below:
- String firstName = rs.getString(“FirstName”);
- String lastName = rs.getString(“LastName”);

```
import java.sql.*;
public class ReadingData2 {

    static Connection con = null;
    static Statement stmt = null;
    static ResultSet rs = null;

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

    private static void readData() throws Exception {
        // Create the SQL
        String sql = "SELECT firstname FROM CUSTOMERS";
        // Get the connection using our utils.
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
        Connection con = DriverManager.getConnection(url, "root",
"");
```

```
if (con != null) {
            stmt = con.createStatement();
            rs = stmt.executeQuery(sql);
            while (rs.next()) {
                String firstName = rs.getString(1);
                 System.out.println("Firstname::"+firstName);
            }
        } else {
            System.out.println("Unable to get the connection");
        }
    }

}
```

UPDATE

```
import java.sql.*;
import javax.swing.JOptionPane;

public class UpdateRecord {
   static Connection con = null;
    static Statement stmt = null;
    static ResultSet rs = null;
    public static void main(String args[]) throws Exception {
        UpdateData();
    }

    private static void UpdateData() throws Exception {
        String sql = "UPDATE customers SET firstname = 'ramesh'
where firstname='ram'";
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
        con = DriverManager.getConnection(url, "root", "");
```

```
if (con != null) {
            stmt = con.createStatement();
            int result=stmt.executeUpdate(sql);
            if(result!=-1){
                JOptionPane.showMessageDialog(null, "Update
record Successfully");
            }else{
                JOptionPane.showMessageDialog(null, "Failed
to update record ");
            }

        } else {
            System.out.println("Unable to get the
connection");
        }
    }
}
```

DELETE

```
import java.sql.*;
import javax.swing.JOptionPane;

public class DeleteRecord {
    static Connection con = null;
    static Statement stmt = null;

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

    private static void DeleteData() throws Exception {
        String sql = "DELETE FROM customers where firstname='ram'";
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost:3306/MyDB";
         con = DriverManager.getConnection(url, "root", "");
```

```
 if (con != null) {
            stmt = con.createStatement();
            int result=stmt.executeUpdate(sql);
            if(result!=-1){
                System.out.println("deleted");
                //JOptionPane.showMessageDialog(null,
"Delete record Successfully");
            }else{
                JOptionPane.showMessageDialog(null, "Failed
to Delete record ");
            }

        } else {
            System.out.println("Unable to get the
connection");
        }
    }
```

PreparedStatement

- This class replaces the old Statement class that we used for executing the queries in the previous examples.
- The usage of this class is pretty simple. It involves three simple steps as listed below:
  1. Create a PreparedStatement
  2. Populate the statement
  3. Execute the prepared statement.
- The basic idea with this class is using the ‘?’ symbol in the query where ever the data is to be substituted and then when the data is available, replace the symbols with the data.
- Step1: To create a prepared statement we use the following method on the Connection object as shown below:
- PreparedStatement prepareStatement (String sql);

- String sql = “INSERT INTO CUSTOMERS VALUE (?, ?, ?, ?, ?, ?, ?, ?)”;
- Since we need to insert data into eight columns, we used eight question marks.
- PreparedStatement stmt = conn. prepareStatement (sql);
- Step 2: Now that we created a prepared statement, its time to populate it with the data. To do this, we use the set methods as shown below:
- stmt.setString (1, name data);
- stmt.setInt (3, ssn data);
- Looks like the above set methods are familiar, right? These work opposite to the get methods we used with ResultSet class couple pages back. The first parameter to the set method denotes the position of ‘?’ to replace and the second argument is the actual data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

public class PrepareStatementDemo {
    static Connection con = null;
    static PreparedStatement pstmt = null;
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            String url = "jdbc:mysql://localhost:3306/MyDB";
            con =
DriverManager.getConnection(url,"root","");
            String query = "INSERT INTO CUSTOMERS
VALUES (?,?,?,?,?,?,?,?)";
            pstmt = con.prepareStatement(query);
            pstmt.setString(1, "harry");
```

```
  pstmt.setString(2, “B");
  pstmt.setString(3, “Porter");
  pstmt.setInt(4, 20);
  pstmt.setInt(5, 123);
  pstmt.setString(6, “ktm");
  pstmt.setString(7, “3");
  pstmt.setString(8, “nepal");
  int result=pstmt.executeUpdate();
if(result!=-1){
System.out.println(“Inserted successfully”);
}else{
System.out.println(“Fail to insert”);
}
```

```
            pstmt.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

- Following are some of the examples that demonstrate the usage of PreparedStatement class.

String sql = “SELECT * FROM CUSTOMER WHERE FIRSTNAME = ? “;

- stmt. setString (1, “John”);

String sql = “UPDATE CUSTOMERS SET FIRSTNAME= ? WHERE SSN = ?”;

- stmt.setString (1, “Smith”);
- stmt.setInt (2, 87690); // Used setInt since ssn is a number in the table

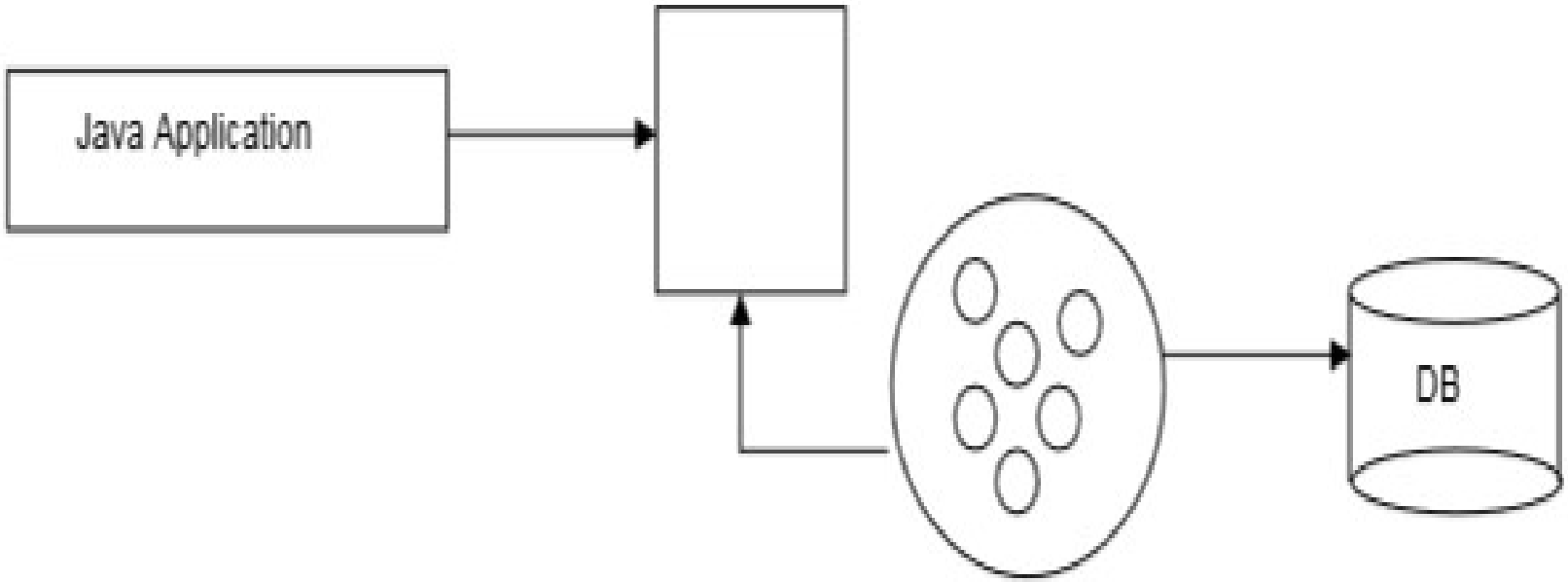
Connection Pooling

- In all the programs we wrote thus far, there is one thing that we are repeatedly doing all the times. Opening a connection at the beginning and closing the connection at the end when everything is done.
- There are two major problems that hunt us when we write database programs.
- Problem 1: Usually, every database has limited number of connections. Let’s say a database supports only 10 connections, and if your program doesn't close the connection every time it ran, the 11th time you run, database will refuse to give you a connection. If you don't close a connection after using it, from database point of view somebody is still using the connection, and it will not reuse this connection until the connection is closed. You need to shutdown the database and restart it to release all the connections which is not good.

- Problem 2: Let's say we have a database that can support a maximum of 200 connections at a time and the application we wrote supports 1000 customers at a time. So, if the 200 connections are used for 200 customers, what will happen to the remaining 800 customers? They are simply kicked off. This is no good because you are loosing business.
- Solution: Need to have a sophisticated mechanism to manage the connections effectively and efficiently by optimizing the usage of database connections.
- Such a mechanism is what we call as connection pooling.
- Now the question is who implements connection pooling? To implement the connection pooling we need to know all the intricacies of database connection. Who else other than the database vendors know better about connection details? This is the reason why database vendors themselves implement connection pooling mechanism in Java using certain

- The connection pooling program comes along when we download the drivers for a particular database. Connection pooling mechanism addresses both the above problems.
- The way the connection pooling works is,
- It initializes a pool of connections with the database.
- When a connection is requested by the application, it returns a connection from the pool.
- The application after using the connection returns it back to the pool.

Connection Pool Program



Connection Pool

- Connection pooling will be implemented in such a way that all the details are hidden behind the scenes.
- The only change we as application developers need to make is the way we retrieve the connection object. Once we get a connection object, the rest of the program will be the same.
- Connection pooling is usually used with large scale enterprise applications where thousands of processes need to access the database at the same time.
- In such situations the application server will be configured to use the connection pooling.