JDBC

Java Data Base Connectivity

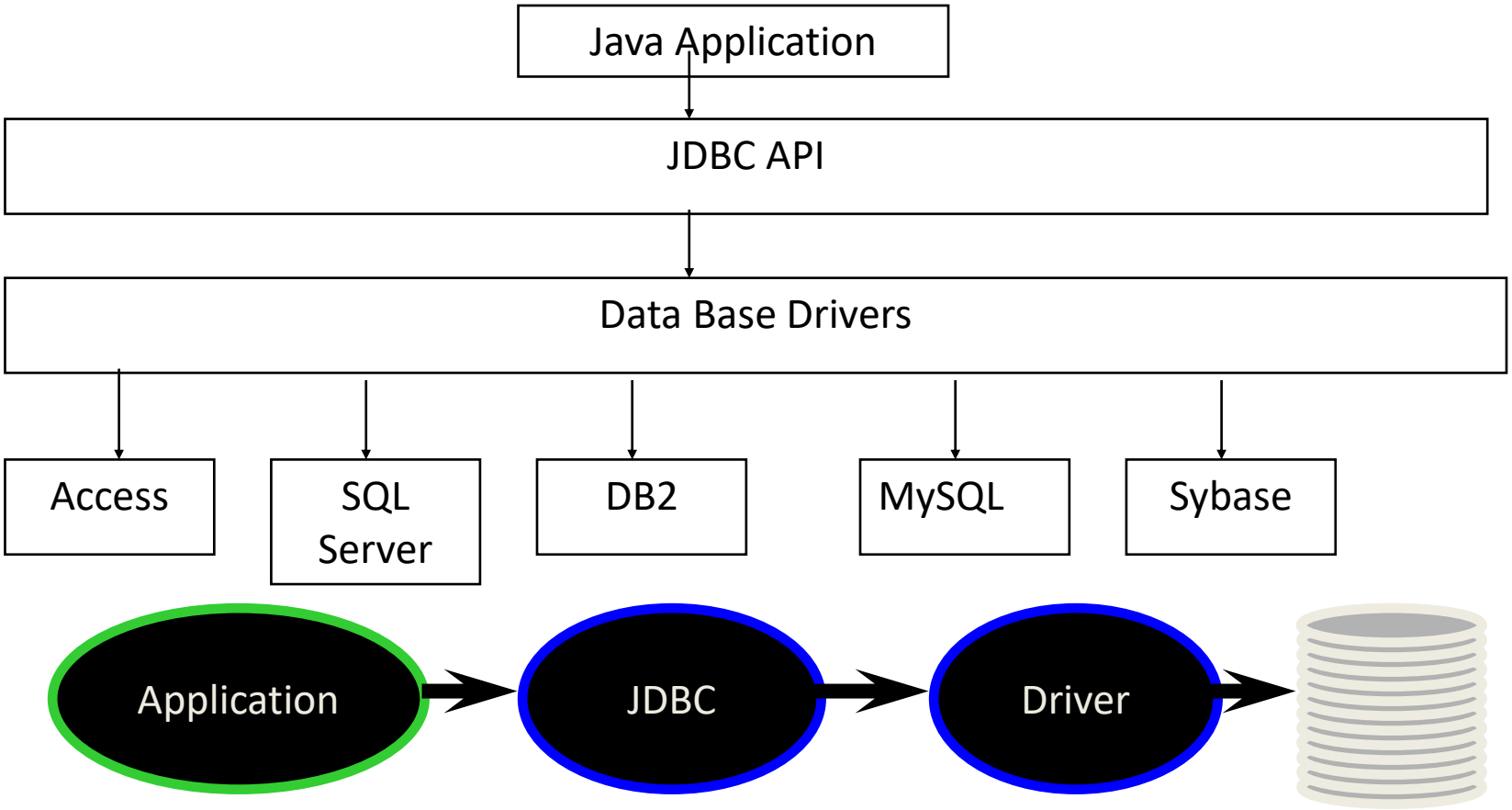
UNIT I

JDBC

- JDBC is a standard interface for connecting to relational databases from Java
- The JDBC Classes and Interfaces are in the ***java.sql*** package
- JDBC is Java API for executing SQL statements
  - Provides a standard API for tool/database developers
  - Possible to write database applications using a pure Java API
  - Easy to send SQL statements to virtually any relational database

- What does JDBC do?
  - Establish a connection with a database
  - Send SQL statements
  - Process the results

JDBC Architecture

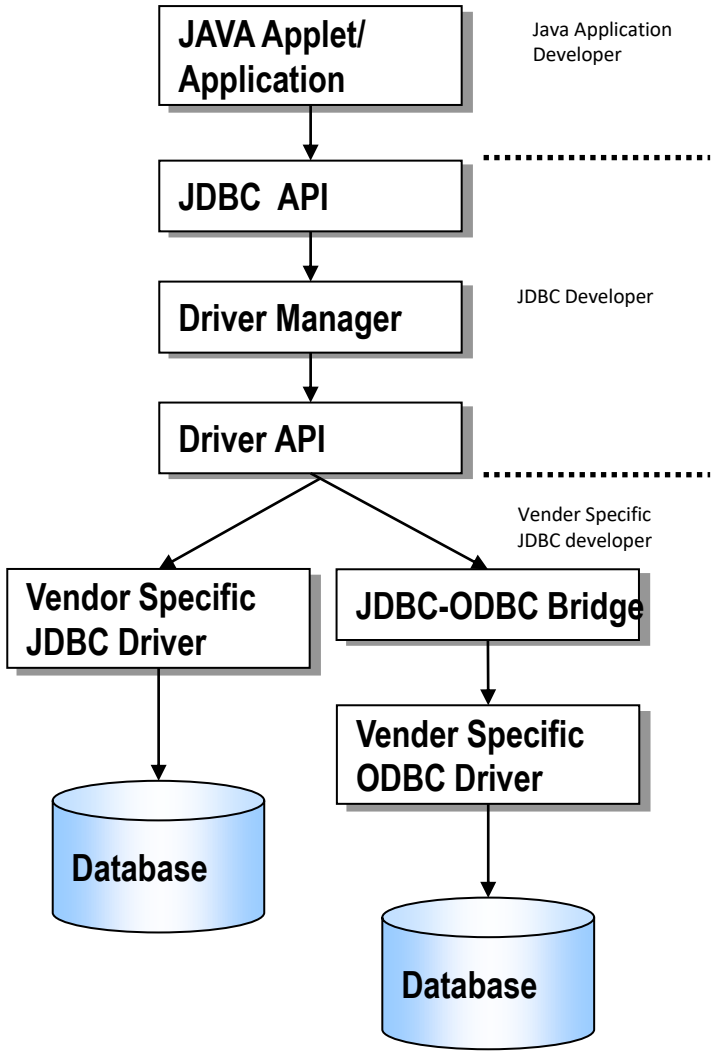


Java code calls JDBC library

JDBC loads a *driver*

Driver talks to a particular database

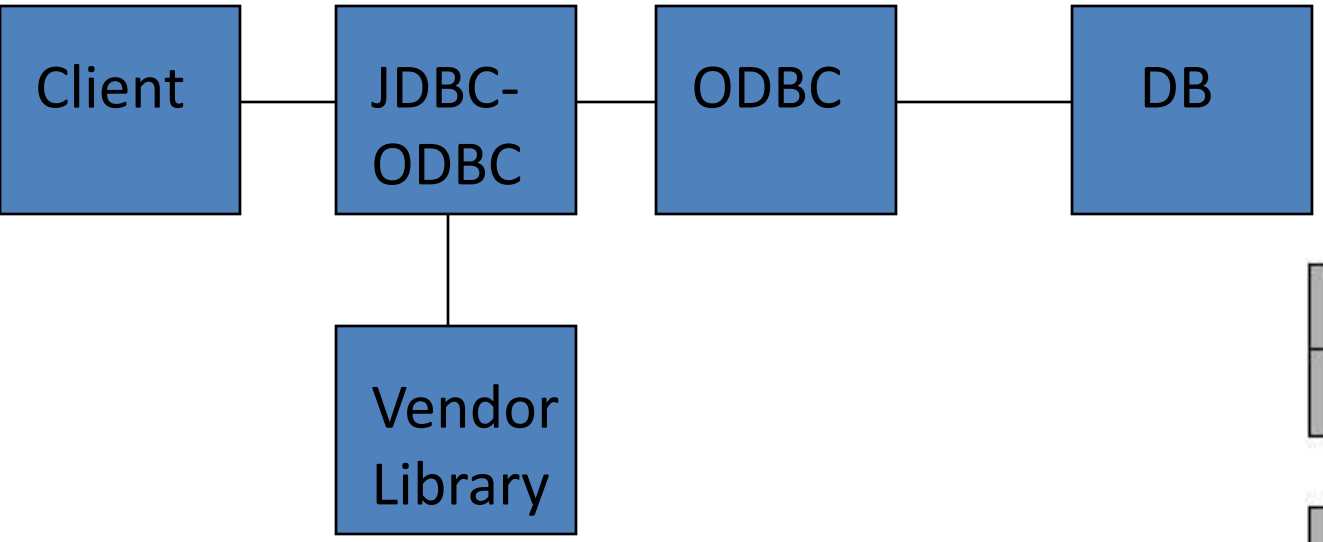
JDBC Model

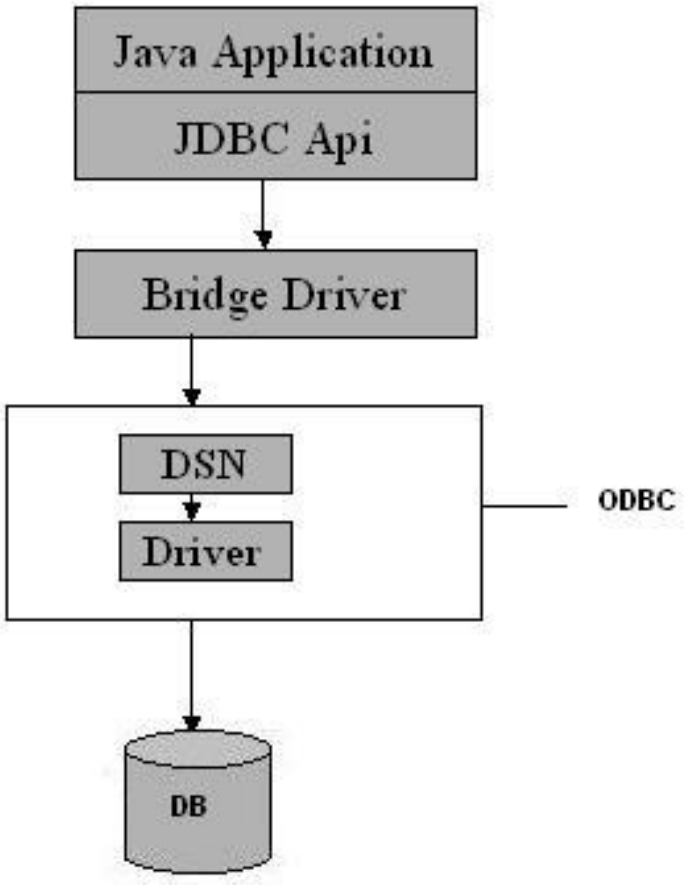

JAVA Applet/ Application
Java Application Developer
JDBC API
Driver Manager
JDBC Developer
Driver API
Vender Specific JDBC developer
Vendor Specific JDBC Driver
JDBC-ODBC Bridge
Vender Specific ODBC Driver
Database
Database

Type 1 Drivers- JDBC ODBC Driver

- Type 1 driver translates all JDBC calls into ODBC calls and send them to ODBC driver.
- The bridge Is usually used when there is no pure – Java driver available for a particular database.
- The JDBC-ODBC bridge drivers are recommended when no other alternative drivers are available.
- The driver is implemented in the sun.jdbc.odbc.JdbcOdbcDriver class.
- The driver is platform dependant as it makes use of ODBC which in turn depends on native libraries of the Operating System.

Type1 Driver- JDBC ODBC Bridge





Type I Driver – JDBC ODBC BRIDGE

- **Advantage:**
  - The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

- **Disadvantage:**

  a). Type 1 Drivers are not portable, since the Bridge driver is not fully written in Java.

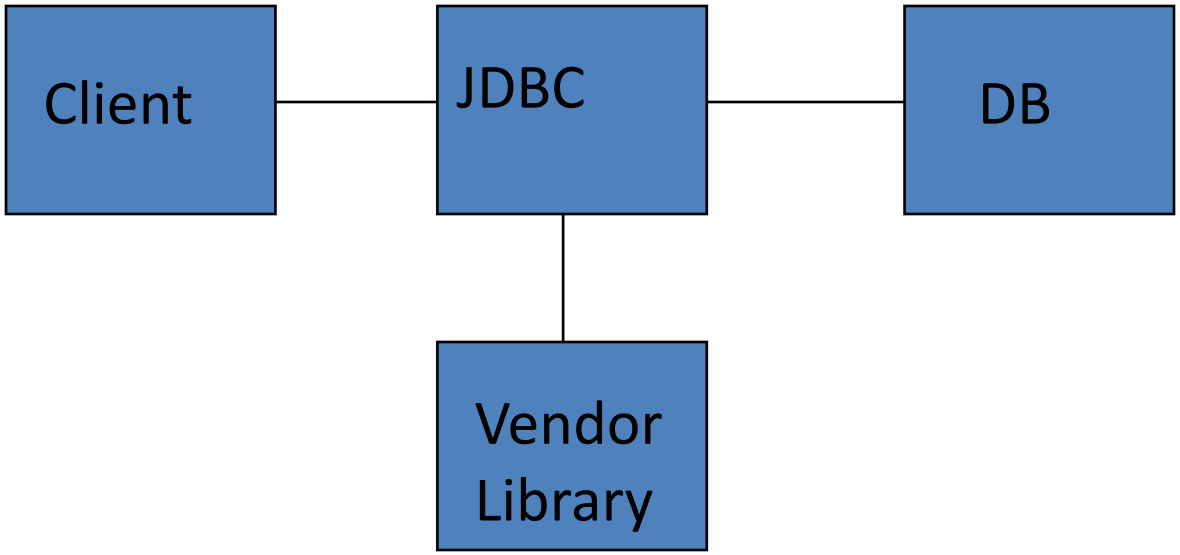
  b). Performance overhead since JDBC calls goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process.

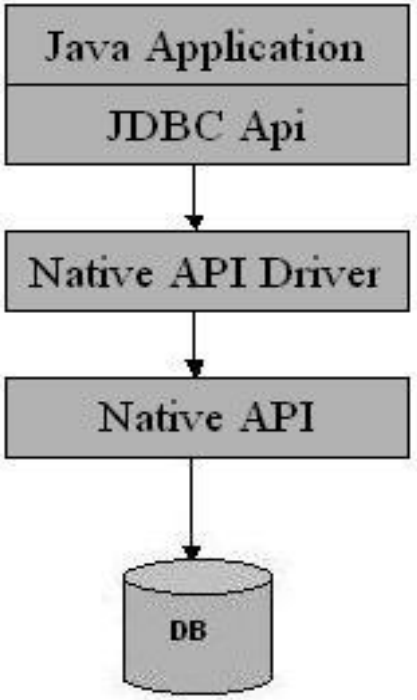
  c). The ODBC driver needs to be installed on the client machine.

  d). Not good for Web.

Type2 Driver- Native API, Partially Java

- Converts JDBC to data base vendors native SQL calls
  - like Type 1 drivers; requires installation of binaries on each client





Type 2 Driver – The Native – API Driver / Partly Java Driver

- The type 2 driver converts JDBC calls into database-specific calls
  - i.e., this driver is specific to a particular database.
  - For example the oracle will have oracle native API.

- The type 2 driver is not written entirely in Java.
- However the type 2 driver provides more functionality and performance than the type 1 driver as it does not have the overhead of the additional ODBC function calls.

- 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.

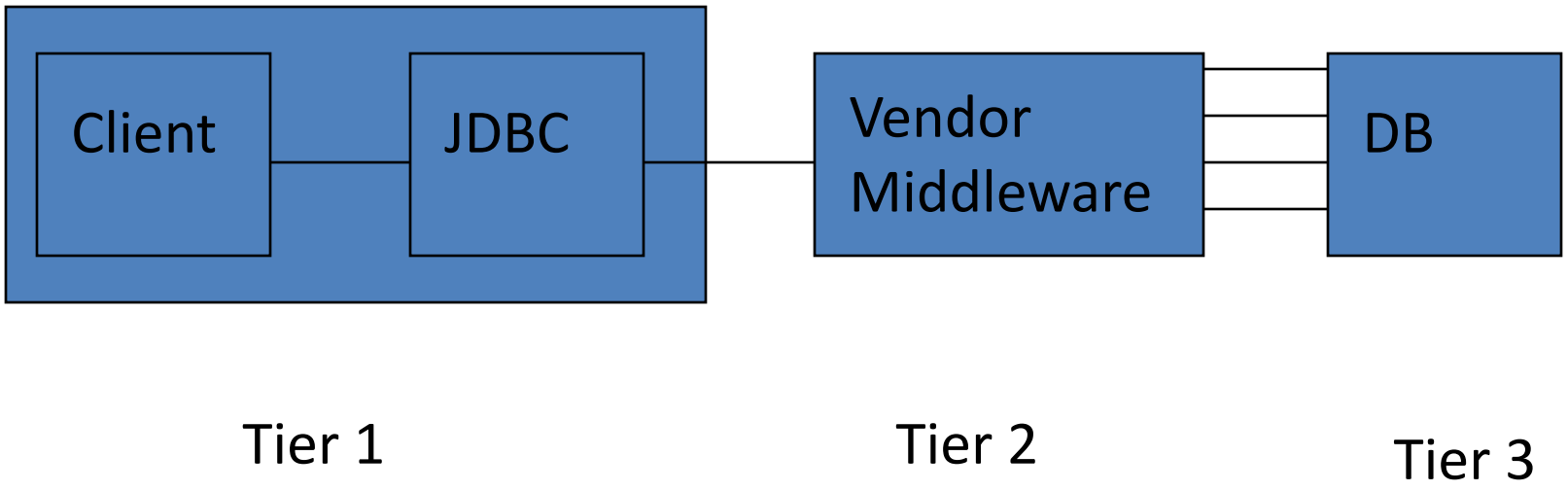
Type 2 Driver

- **Advantage:**
  - Better performance than Type 1 since no JDBC to ODBC translation is needed.

- **Disadvantage:**
  - Native API must be installed in the Client System and hence Type 2 Drivers cannot be used for internet.
  - Type 2 Drivers are not written in the Java Language which forms the portability issues.
  - If we change the Database we have to change the native API as it is specific to the database.
  - Not safe for Threads.

Type 3 Driver-JDBC Network Driver, partially java

- Translates JDBC to a DBMS independent network protocol
- Typically talks directly with a middleware product which in turn talks to the RDBMS
  - Jaguar, DBAnywhere, SequeLink
- Most flexible driver type all java



Type 3: All Java/ Net-Protocol Driver

Advantages:

a) Since the communication between client and the middleware server is database independent, there is no need for the vendor database library on the client machine.

b) The middleware server can provide typical middleware services like caching (connections, query results, and so on), load balancing etc.,

c) This driver is fully written in Java and hence portable.

d) This driver is very flexible allows access to multiple databases using one driver.

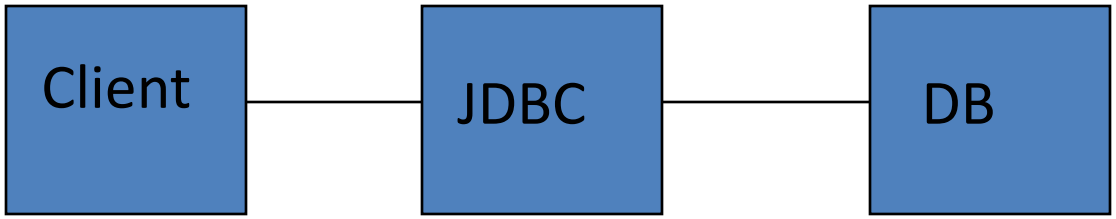
f) Type 3 driver Is more efficient amongst all driver types.

Disadvantage:

Requires another server application to install and maintain.

Type 4 Driver- All Java

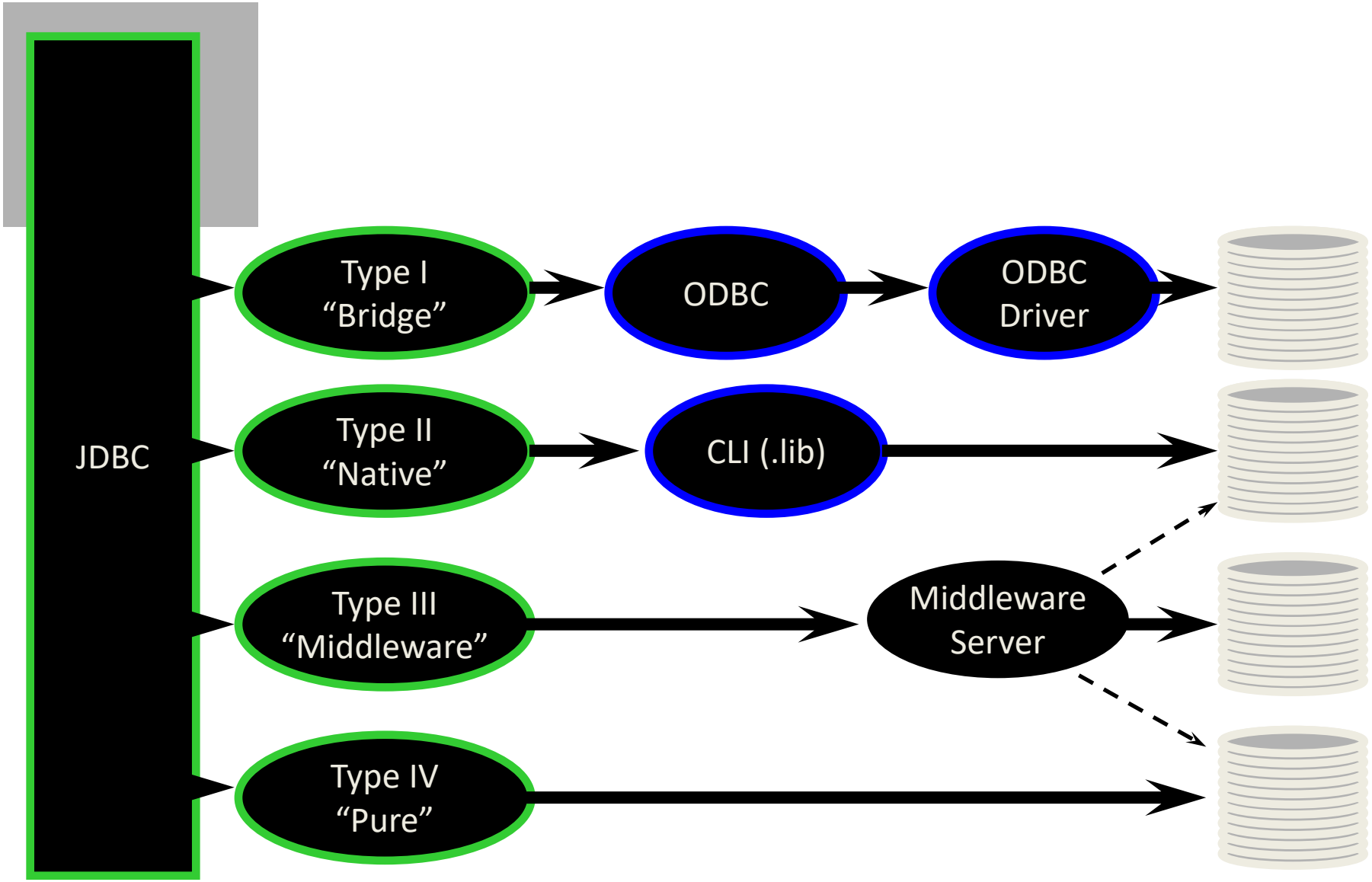
- Converts JDBC directly to native API used by the RDBMS
- Compiles into the application , applet or servlet; doesn't require anything to be installed on client machine, except JVM
- Handiest driver type



Type 4 Driver- All Java

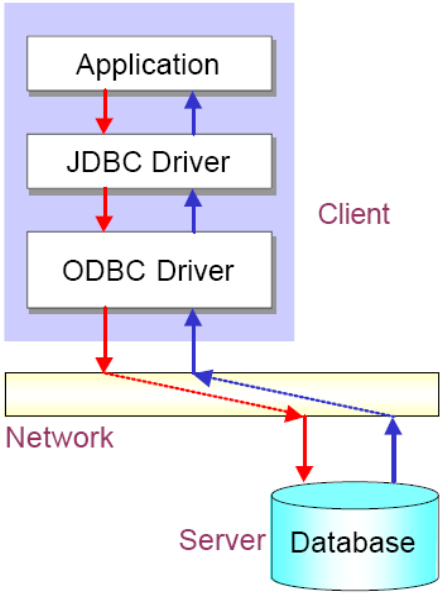
- **Advantage:**
  - Web application mainly uses this driver.
  - Number of translation layers is very less.
  - You do not need to install special software on the client or server.
- **Disadvantage:**
  - At client side, a separate driver is needed for each database.

JDBC - DRIVERS

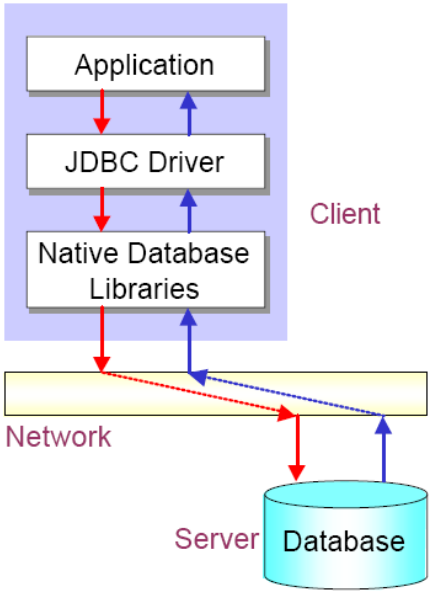


JDBC Driver Types

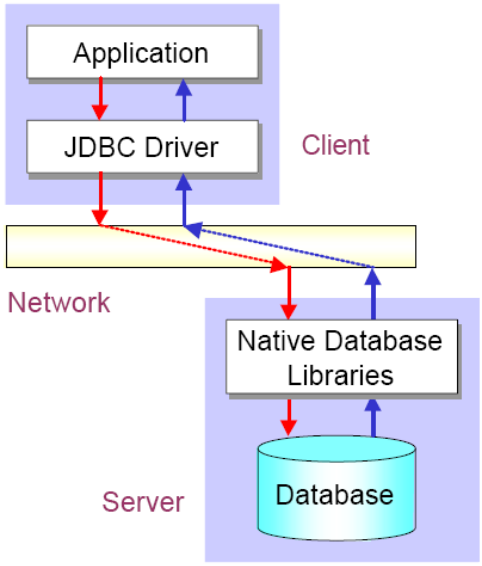
- Type 1 (Bridge)
  - JDBC-ODBC Bridge (The JDBC-ODBC Bridge is an example of a Type 1 driver.)
- Type 2 (Native)
  - Native API, partially java (Oracle's OCI (Oracle Call Interface) client-side driver is an example of a Type 2 driver.)
- Type 3 (Middleware)
  - JDBC Network Driver, partially java
- Type 4 (Pure)
  - 100% Java



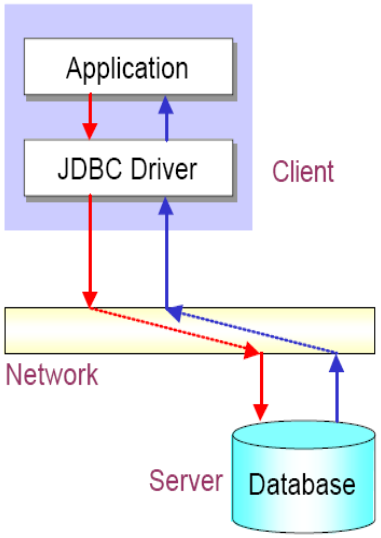
Type I : JDBC-ODBC bridge + ODBC



Type II : Native API partly Java



Type III : Java Net pure Java



Type IV : Native protocol pure Java

JDBC Process

- The JDBC process is divided in to five routines. These includes.

  1.Loading the JDBC driver.

  2.Connecting to the DBMS.

  3.Creating and executing a SQL statement.

  4.Processing data returned by the DBMS.

  5.Terminating the connection/Close the connection.

1. Loading the JDBC driver

- The jdbc driver must be loaded before connecting to the DBMS.
- The Class.for Name() method is used to load the JDBC driver
- **sun.jdbc.odbc.jdbc.odbcDriver** to load the JDBC/odbc Bridge driver as shown below:

Class.forName("Type of the Driver");

- **Ex:**
  - **JDBC-ODBC: sun.jdbc.odbc.JdbcOdbcDriver**
  - **Oracle driver: oracle.jdbc.driver.OracleDriver**
  - **MySQL: com.mysql.jdbc.Driver**

2.Connecting to the DBMS

- Once the driver is loaded, the J2EE component must connect to the DBMS using the **DriverManger.getConnection()** method.

- The **DriverManager.getConnection()** method is passed the URL of the database and the user Id and Password if required by the DBMS.

- The DriverManager.getConnection() method returns the connection interface that is used throughout the process to reference the data base.

Example of Connecting

```
String url=”jdbc:odbc:EmployeeInformation”;
 Strting user ID=””;
 String password=””;
Connection con;
 try
  {
    Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
    Con =
    DriverManager.getConnection(url,userID,password);
  }
```

3. Creating and executing a SQL statement

- Next step after the JDBC driver is loaded and connection is successfully made with a particular database managed by the DBMS,is to send a SQL query to the DBMS for processing.A SQL query consists ofa a series of SQL commands that direct the DBMS to do something such has to return rows data to the J2EE components.
-
- createStatement() method of Connection is used to create a Statement object.The Statement object is used to execute query and return a ResultSet object that contain the response from the DBMS which is usually one or more rows of information requested by the J2EE component.

```
Resultset result;
      String url=”jdbc:odbc:EmployeeInformation”;
      Strting user ID=””;
   String password=””;
  Connection con;
  try
   {
      Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
      Connection con;
          con = DriverManager.getConnection(url,userID,password);
          Statement stat = con.createStatement();
          Result = stat.executeQuery(“select * from emp”);
   }
```

4. Process Data Returned by the DBMS

- The result returned by the DBMS is already assigned to the ResultSet object.
- The first time that the next() method of the ResultSet is called, the ResultSet pointer is positioned at the first row in the ResultSet
- It returns a Boolean value true or false
  - If it is false , then it indicates that no rows are present in the ResultSet.
  - If it is true, then it indicates at least one row of data is present in the ResultSet.
- The getString() method of the ResultSet object is used to copy the value of a specified column if in the current row of the ResultSet to a String object.

5.Terminate the Connection to the DBMS

- The Connection to DBMS is terminated by using the close() method of the Connection object once the J2EE component is finished accessing the DBMs.
- The Close() method throws an exception.
- Closing of the database connection automatically close the resultset.

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

Process

- forName()
- getConnection()
- Statement interface
- Qurey sending and retrieve
- Process the results

Example program

emp

Eno	Ename
101	Kumar
102	Harsha

```
import java.sql.*;
public class JavaApplication12 {
   public static void main(String args[])
   {
      try
            {
     Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
      Connection con=DriverManager.getConnection("jdbc:ucanaccess://C:\\e\\Emp.accdb");
     Statement st=con.createStatement();

     ResultSet rs=st.executeQuery("select * from emp");
     while(rs.next())
                        {
          System.out.println("emp Number is:"+rs.getInt(1));
           System.out.println("emp name is:"+rs.getString(2));
                        }
          }
     catch(Exception e)
           {
        System.out.println(e.getMessage());
          }
   } }
```

Importing package

Loading the Driver

Statement object

Connection establishment

Sending and retrieved query

Processing the Results

Statement Objects

- Once a connection to the database is opened, the J2EE component sends a query to access data contained in the database.
- One of the three types of Statement objects is used to execute the query. These objects are:
- Statement
  - Which executes a query immediately
- PreparedStatement
  - Which is used to execute a compiled query
- CallableStatement
  - Which is used to execute store procedures

Statement interfaces

- Statement
- PreparedStatement
- CallableStatement

1. The Statement Object

- The Statement object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled.

- The Statement object contains the executeQuery("query") method, which is passed the query as an argument.

- The query is then transmitted to the DBMS for processing.

1. The Statement Object contd..

- executeQuery() method
  - returns one ResultSet object that contains rows, columns, and metadata that represent data requested by query.
- execute() method
  - is used when there may be multiple results returned.
- executeUpdate() method
  - is used to execute queries that contain UPDATE and DELETE SQL statements, which changes values in a row and removes a row respectively.
- The executeUpdate() is used to
  - INSERT
  - UPDATE
  - DELETE and
  - DDL statements

Examples

```
import java.sql.*;
class A
{
public static void main(String args[])
{
Statement s;
ResultSet rs;
Connection con;
try{
      Class.forName(“”sun.jdbc,odbc.JdbcOdbcDriver”);
con  = DriverManager.getConnection(jdbc:odbc:dsn,”,”);
      String query = “SELECT * FROM emp”;
      s = con.createStatement();
      rs = s.executeQuery(query);
      while(rs.next()){
            //WRITE THE CODE TO RETRIEVE THE DATA
      s.close();
      con.close();
}catch(Exception error) {
      System.err.println(“Exception occurred”);
}
```

Select query

```
Statement statement = connection.createStatement();
String sql = "select * from people";
ResultSet result = statement.executeQuery(sql);
```

Insert query

```
Statement statement = connection.createStatement();
String sql = "INSERT INTO tablename VALUES (1,2,’kumar’)";
int rowsAffected= stmt.executeUpdate(sql);
```

update query

```
Statement statement = connection.createStatement(); String
sql = "update tablename set name=‘hari' where id=123";
int rowsAffected = statement.executeUpdate(sql);
```

Delete query

```
Statement statement = connection.createStatement(); String
sql = "DELETE FROM tablename WHERE id = 101";
int rowsAffected =stmt.executeUpdate(sql);
```

Reading data from keyboard and inserting into the table

```
Scanner br=new Scanner(System.in);

System.out.print("Enter student number: ");
int eno=br.nextInt();

System.out.print("Enter student name: ");
String ename=br.nextLine();

System.out.print("Enter Salary: ");
int esal=br.nextInt();

Statement statement = connection.createStatement();

String query="insert into employee values("+eno+",' "+ename+" ', "+esal+")";
int rowsAffected = statement.executeUpdate(query);
```

2. PreparedStatement Object

- A SQL query must be compiled before the DBMS processes the query.
- A SQL query can be precompiled and executed by using the PreparedStatement
- A question mark is used as a placeholder for a value that is inserted into the query after the query is compiled.
  - It is this value that changes each time the query is executed.
- The prepareStatement() method of the Connection object is called to return the PreparedStatement object.
  - **The prepareStatement() method is passed the query, which is then precompiled.**

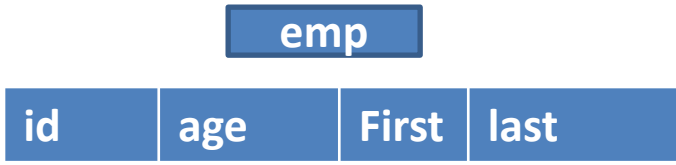
2. PreparedStatement Object contd..

- There are set and get methods available to set and get the details of the fields in the table
  - The syntax is setxxx() and getxxx()
  - The setxxx(int, datatype) requires two parameters
  - the first parameter is an integer that identifies the position of the question mark placeholder
  - the second parameter is the value that replaces the question mark placeholder.

Example

```
import java.sql.*;
class A
{
public static void main(String args[])
{
Statement s;
ResultSet rs;
Connection con;
try{
  Class.forName(“Drivername”);
con  = DriverManager.getConnection(connectionstring,username,password);
String query = “SELECT * FROM emp where eno=?”;
PreparedStatement pstatement = con.prepareStatement(query);
pstatement.setInt(1,123);
rs = pstatement.executeQuery();
while(rs.next()){
                //WRITE THE CODE TO RETRIEVE THE DATA
  s.close();
  con.close();
}catch(Exception error) {System.err.println(“Exception occurred”);
}
```

Example2



- Retrieve the data using the select query
- Update the table based on the age for a given id
  - String sql = "UPDATE emp set age=? WHERE id=?";
  - stmt = conn.prepareStatement(sql);
  - stmt.setInt(1, 35);
  - stmt.setInt(2, 102);
- Delete the employee based on a given id
  - delete from emp where id=?"

CallableStatement

- The CallableStatement object is used to call a stored procedures from within a J2EE object.
- A stored procedure is a block of code and is identified by a unique name.
- The type and style of code depends on the DBMS vendor and can be written in PL/SQL, Transact-SQL, C, or another programming language.

CallableStatement contd..

- The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are:-
- IN
  - It contains any data that needs to be passed to the stored procedure and whose value is assigned using the setxxx() method as described.
- OUT
  - It contains the value returned by the stored procedures if any. It must be registered using the registerOut() method and then is later received by the J2EE component using the getxxx() method.
- INOUT.
  - It is a single parameter that is used to both pass information to the stored procedure and retrieve information from a stored procedure using the techniques described.

- the preparedCall() method of the connection object is called and is passed the query.
- The registerOutParameter() method requires two parameters.
- The first parameter is an integer that represents the number of the parameter.
- The second parameter is the datatype of the value returned by the stored procedure.

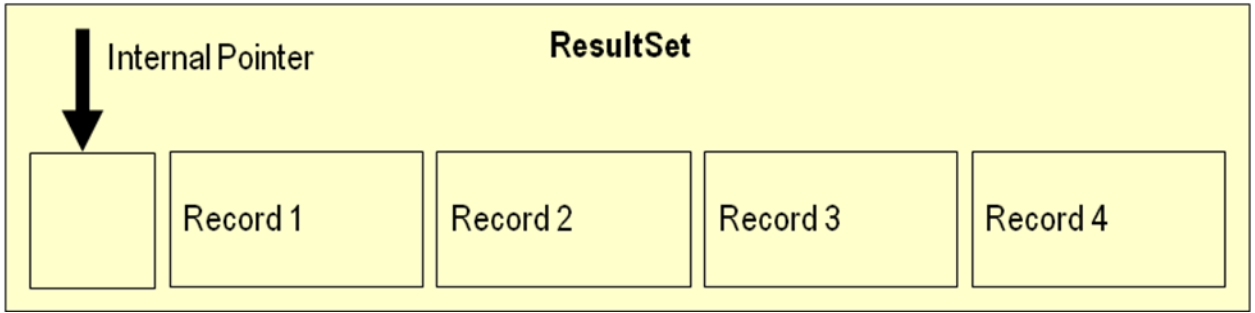
```
class A
{public static void main(String args[])
{Statement s;
ResultSet rs;
Connection con;
try{
Class.forName(“sun.jdbc,odbc.JdbcOdbcDriver”);
Con=DriverManager.getConnection(jdbc:odbc:
dsn,”,”);
}
catch(ClassNotFoundException error)
{
    System.err.println(”Unable to load the JDBC/ODBC bridge.”+error);
}
```

```
try{

String query="{ CALL LastOrderNumber(?) }";
CallableStatement cstatement = con.prepareCall(query);
cstatement.registerOutParameter(1,Types.VARCHAR);
    cstatement.execute();
    lastOrderNumber = cstatement.getString(1);
            while(rs.next()){
            //code to disply the results
            }
            s.close();
            con.close();
            }
catch(SQLException error) {
    System.err.println("SQL error."+error);
}
```

The ResultSet Object

- JDBC returns the results of a query in a ResultSet object
  - ResultSet object contains all of the rows which satisfied the conditions in an SQL statement
- A ResultSet object maintains a cursor pointing to its current row of data
  - Use next() to step through the result set row by row
  - next() returns TRUE if there are still remaining records
  - getString(), getInt(), and getXXX() assign each value to a Java variable



Accessing rows returned from a query

- Rows produced by an *executeQuery()* will be returned within an object of type *ResultSet*, which is of use in acting on it later.
- The *executeUpdate()* method returns an int that represents the number of rows updated by the SQL statement.
- Rows are retrieved from a *ResultSet* object by iterating through each of the rows and requesting information on the contents of each column. Iteration through its rows is similar to iterating through a set of elements in an *Enumeration* object.

Result Set contd..

- The *ResultSet* class provides a cursor (also called handle) that can be used to access each row.
- The method *next*() of the *ResultSet* class is used to move the cursor that points to a row in the *ResultSet*.
- Once the cursor is positioned on a row, we can request the data for each of its columns.
- Initially the cursor is set just before the first row.
- The first invocation of the method *next()* sets the cursor and first row.
- That is to access the first row in the *result set*, the method *next* must be called once.
- Each invocation of the next() makes the cursor move to the next row.
- The method *next()* returns a boolean value based on whether the next row is present.
  - If a row exists, it returns true otherwise it, returns false.

Result Set

- **Accessing column data**
  - The method getXXX() is used to access the columns in a row that are being pointed by the cursor currently.
  - Here XXX represents the java datatype as each column may contain a different datatype, we need a *ResultSet* method specific to that datatype.
  - The method *getInt(int column)* can be used to retrieve a column with an integer value
  - *getString(int column)* can be used to retrieve a column with a string value.
  - In General, the *getString(int column)* method is used to get all types of database value.
    - In such cases the retrieved values will be converted to a string type in java.

Example

```
Import java.sql.*;
Public class readdata
{
Public static viod main(String arg[])
{
resultSet result;
string url=”jdbc:odbc:Employee Information”;
string userID=” “;
string password=” “;
try
{
Class.forName(sun.jdbc.odbc.jdbcodbcDriver”);
Connection con;
Con=DriverMAnager.getConnection(url,userID,password);
Statement stat=con.createStatement();
Result=stat.executeQuery(“select * from emp”);
While(result.next())
{
String row=result.getstring(1)=”
    “+result.getstring(2)+”
    “+result.getstring(3);
System.out.println(row) ;
}
}
Catch(Exception e)
{
System.out.println(“error:”+e);
}
}
}
```