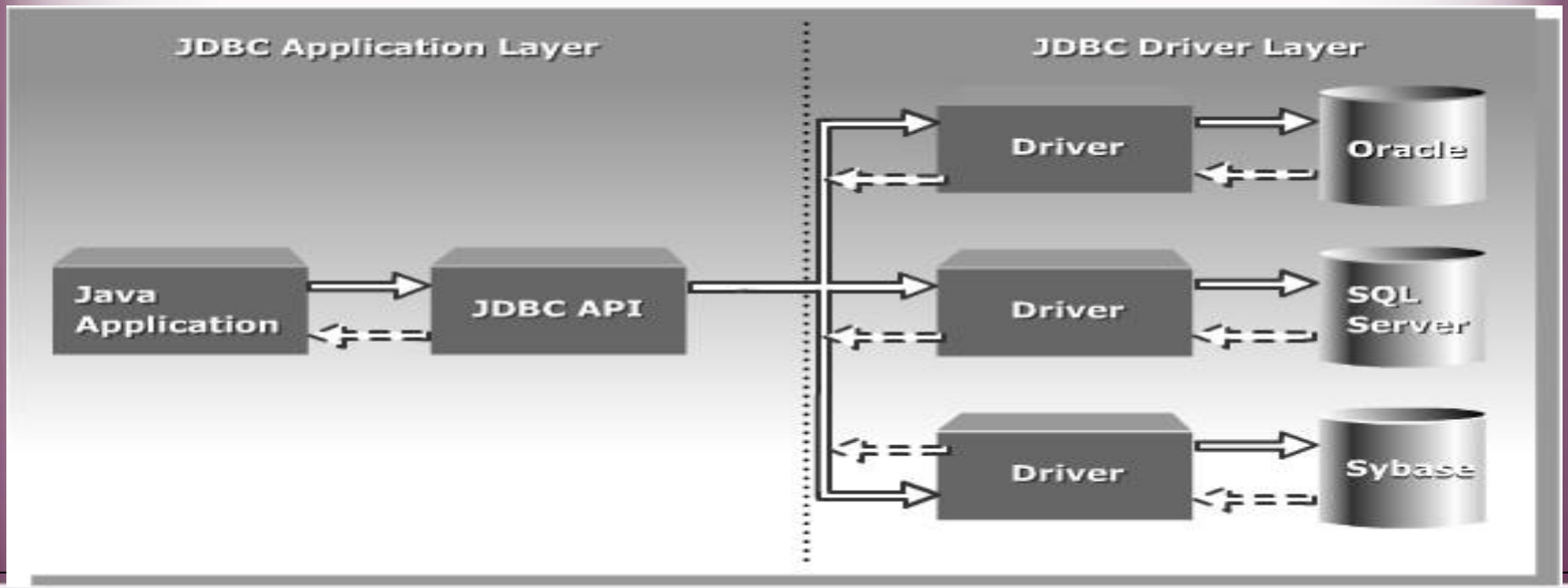
JDBC

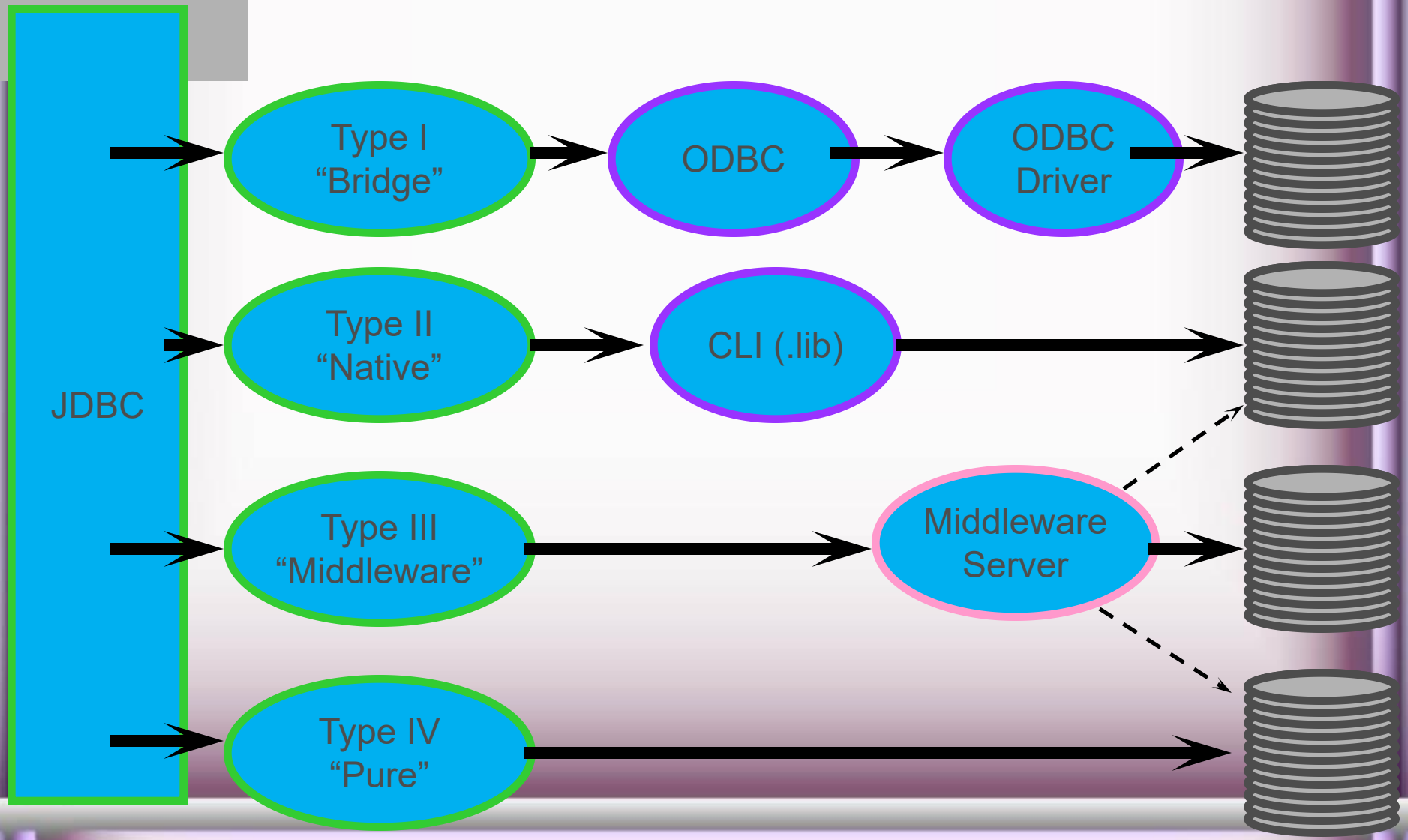
- JDBC is a standard interface for connecting to relational databases from Java.
- The JDBC classes and interfaces are in the java.sql package.
- JDBC 1.22 is part of JDK 1.1; JDBC 2.0 is part of Java 2

Database Connectivity

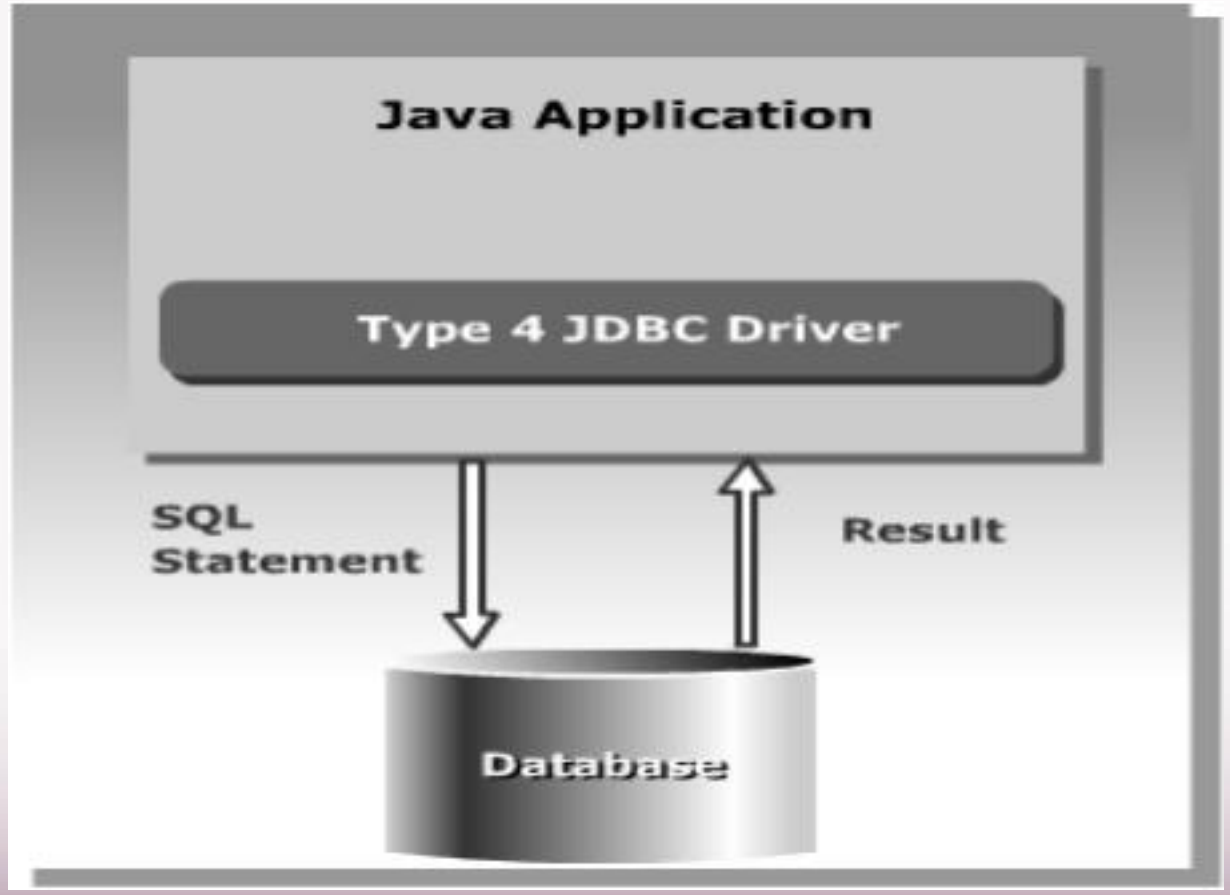
- JDBC Architecture:
  - Provides the mechanism to translate Java statements into SQL statements.
  - Can be classified into two layers:
    - JDBC application layer
    - JDBC driver layer



JDBC Drivers



Type -4 Native Protocol Pure Java Driver



Type -4 Native Protocol Pure Java Driver

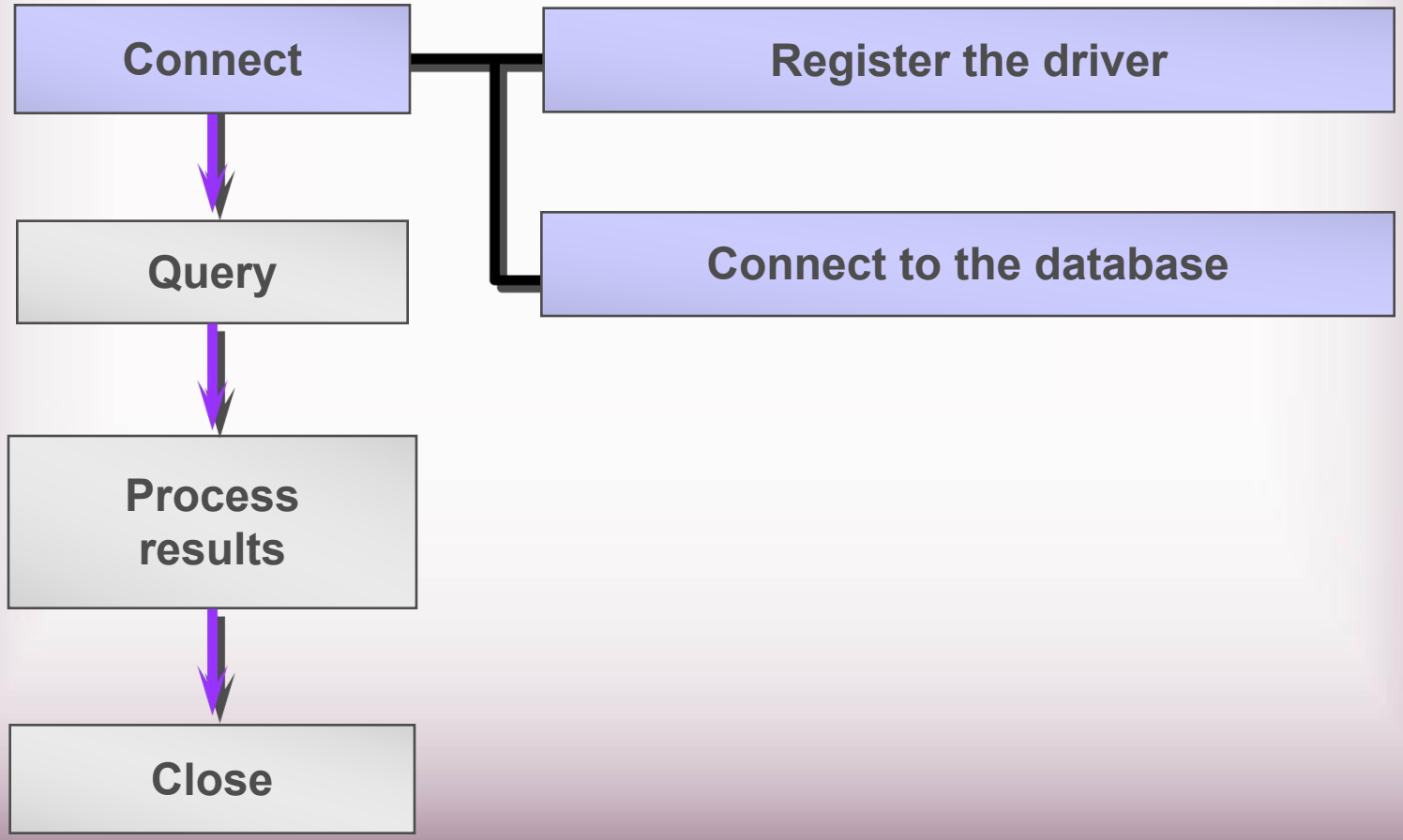
- **Advantages**
  - They Don't translate the requests into an intermediary format (such as ODBC),
  - No need for a middleware layer to service requests.
  - The JVM can manage all aspects of the application-to-database connection facilitating debugging.
- **Disadvantages**
  - Clients require a separate driver for each database.
  - Drivers are database dependent.

Java.sql packages.

The commonly used classes and interfaces in the JDBC API are:

- **DriverManager class**: Loads the driver for a database.
- **Driver interface**: Represents a database driver. All JDBC driver classes must implement the Driver interface.
- **Connection interface**: Enables you to establish a connection between a Java application and a database.
- **Statement interface**: Enables you to execute SQL statements.
- **ResultSet interface**: Represents the information retrieved from a database.
- **SQLException class:** Provides information about the *exceptions* that occur while interacting with databases.

Querying a Database With JDBC



Loading a Driver Programmatically:

- **forName()**
  - In the java.lang.Class class.
  - Loads the JDBC driver and registers the driver with the driver manager.

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

- Required before Java 6
  - **Not required after Java 6 and JDBC 4.0 API**

- *Need to place the Product Specific JAR file with JDBC 4.X driver in class path.*
  - Java automatically detects the Driver class and loads it.

-

Connecting to a Database

- The DriverManager class provides the getConnection() method to create a Connection object.
- The getConnection()method has the form
- Connection getConnection (String <url>, String <username>, String <password>)
- JDBC uses a URL to identify the database connection.

jdbc:<*subprotocol*>:<*subname*>

jdbc:oracle:<*driver*>:@<*database*>

JDBC URLs with Oracle Drivers

- Thin driver

```
jdbc:oracle:thin:@<host>:<port>:<SID>
```

```
Connection conn = DriverManager.getConnection
      ("jdbc:oracle:thin:@myhost:1521:orcl",
      "scott", "tiger");
```

jdbc:microsoft:sqlserver:@10.161.3.94:1433:javatraining,sa,”L2USER;

Data Base Connection Property

- **src/resources/DbConnection.properties**

```
datasource.driver=oracle.jdbc.driver.OracleDriver

datasource.url=jdbc:oracle:thin:@localhost:1521:XE

datasource.user=system

datasource.password=srivatsan
```

Data Base Connection Property

- src/resources/DbConnection.properties

- datasource.url=jdbc:mysql://localhost:3306/test
- datasource.username=root
- datasource.password=srivatsan
- datasource.driverClassName=com.mysql.jdbc.Driver

Connection Utility

```
public static Connection getMySqlConnection() {

Connection con = null;
try {

String fileName = "resources/DbConnection.properties";

Properties props=new Properties();

InputStream  inStream=

DataBaseConnection.class.getClassLoader().
                getResourceAsStream(fileName);
```

Connection Utility

```
props.load(inStream);

con=DriverManager.getConnection(
    props.getProperty("datasource.url"),
   props.getProperty("datasource.username"),
   props.getProperty("datasource.password"));

} catch (SQLException | IOException e) {

e.printStackTrace();
}
    return con;
}
```

Statements- Statement

- *Statement*
  - *Used to execute string-based SQL queries*
  - *Code becomes less readable because of concatenation of SQL strings*
  - **Query is passed with inline values to the database**.
    - Because of which query optimization is not possible
  - **Prevents cache usage**.
  - *Suitable for DDL queries like CREATE, ALTER, and DROP*

Statements - PreparedStatement

- *PreparedStatement* extends the *Statement* interface.
- **Provides methods to bind various object types**
- **Code becomes easy to understand**:
- Uses pre-compilation.
  - As soon as the database gets a query, it will check the cache before pre-compiling the query.
- Used for statements that is called more than once.
- **Suitable for DML queries like Insert,Update and Delete**

Statements - PreparedStatement

```
String sqlAdd = "insert into emp values(?,?,?)";

PreparedStatement pstmt = con.prepareStatement(sqlAdd);

    pstmt.setInt(1, empid);
    pstmt.setString(2, empname);
    pstmt.setDouble(3, salary);

int n = pstmt.executeUpdate();
```

Executing JDBC Statements

- Can Use SQL statements to send requests to a database to retrieve results.
- The Following methods are used to send static SQL statements to a database:
  - **ResultSet executeQuery**(String str)
  - **int executeUpdate**(String str)
  - **boolean execute**(String str)

Dao Interface

```
public interface DataAccess<T> {


public boolean add(T t);

public List<T> findAll();

public boolean remove(T t);

public int update(int key,T t);

public T findById(int key);


}
```

DAO Implementation

```
public class InvoiceDaoImpl implements DataAccess<Invoice>
{

private Connection connection;

public InvoiceService(Connection connection) {
super();
this.connection = connection;
}
```

DAO Implementation

```
public boolean add(Invoice t) {

String sql = "insert into invoice values(?,?,?)";
int rowAdded =0;

try (PreparedStatement pstmt = connection.prepareStatement(sql)) {

pstmt.setInt(1, t.getInvoiceNumber());
pstmt.setString(2, t.getCustomerName());
pstmt.setDouble(3, t.getAmount());
      rowAdded = pstmt.executeUpdate();
} catch (SQLException e) {
       e.printStackTrace();
}
return rowAdded==1?true:false;
}
```

DAO Implementation

```
public List<Invoice> findAll() {
String sql = "select * from  invoice ";

List<Invoice> invoiceList = new ArrayList<>();
        try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {

ResultSet result = pstmt.executeQuery();

while(result.next()) {

int invoiceNumber =result.getInt("invoiceNumber");
String customerName = result.getString("customerName");
double amount = result.getDouble("amount");
```

DAO Implementation

```
Invoice invoice = new Invoice(invoiceNumber, customerName,
amount);

    invoiceList.add(invoice);
}

} catch (SQLException e) {
e.printStackTrace();
}

return invoiceList;
}
```

Local Date to java.sql Date

- ```
  LocalDate date = LocalDate.of(2021, 8, 17);
  ```
-
- ```
      Date sqlDate = Date.valueOf(date);
  ```

DAO Implementation

```
public boolean remove(Invoice t) {

String sql = "delete from invoice where invoiceNumber=?";
      int rowDeleted =0;

try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {
      pstmt.setInt(1, t.getInvoiceNumber());
            rowDeleted = pstmt.executeUpdate();
} catch (SQLException e) {
e.printStackTrace();
}
return rowDeleted==1?true:false;
}
```

DAO Implementation

```
public void closeConnection() {


try {
this.connection.close();
} catch (SQLException e) {
e.printStackTrace();
}
}
```

DAO Implementation

```
public Invoice findById(int key) {

String sql = "select * from invoice where invoiceNumber=?";

    Invoice foundInvoice =null;

try (PreparedStatement pstmt =
connection.prepareStatement(sql)) {

pstmt.setInt(1, key);

ResultSet result = pstmt.executeQuery();
```

DAO Implementation

```
 if(result.next()) {

 int invoiceNumber =result.getInt("invoiceNumber");
String customerName = result.getString("customerName");
double amount = result.getDouble("amount");

foundInvoice = new Invoice(invoiceNumber, customerName,
amount);
 }
} catch (SQLException e) {
e.printStackTrace();
}
return foundInvoice;
}
```

Using Joins

```
public class CourseList {

private String courseName;
private List<Student> studList;

}

public class Student {

private long studentNumber;
private String studentName;
}
```

Using Joins

```
public CourseList queryByCourse(String courseName) throws
    Exception{

String sql="SELECT Mystudent.studentNumber,
    MyStudent.studentName, MyCourse.courseName from
    MyStudent LEFT OUTER JOIN MyCourse on
    Mystudent.coursecode = Mycourse.coursecode where
    mycourse.courseName =?";

 PreparedStatement pstmt = con.prepareStatement(sql);

 pstmt.setString(1, courseName);
```

Using Joins

```
List<Student> studList = new ArrayList<>();

 ResultSet rs = pstmt.executeQuery();

 while(rs.next())
 {
  studList.add(new Student(rs.getLong(1),rs.getString(2)));
  }

 CourseList course = new CourseList();
  course.setCourseName(courseName);
  course.setStudList(studList);
     return course;
}
```

Transaction

- Is a set of one or more SQL statements that are executed as a single unit.
- Is complete only when all the SQL statements in a transaction execute successfully.
- Maintains consistency of data in a database.

Managing Database Transactions

- JDBC API provides support for transaction management.
- The database transactions can be committed in two ways in the JDBC applications:
  - Implicit: The `Connection` object uses the auto-commit mode to execute the SQL statements implicitly.
  - Explicit: The auto-commit mode is set to false to commit the transaction statement explicitly. The method call to set the auto-commit mode to false is:

```
con.setAutoCommit(false);
```

Save Point

- Used to save the current state of the database which can be rolled-back afterwards to that state of the database.
- Savepoints are similar to the SQL Transactions and are generally to rollback if something goes wrong within the current transaction.
- The connection.setSavepoint() method of Connection interface in Java is used to create an object which references a current state of the database within the transaction.
- connection.setSavepoint()

Managing Database Transaction

```
con.setAutoCommit(false);

String sqlOne ="insert into student values "+"(101 ,'Navven',87)";

        PreparedStatement pstmt1  = con.prepareStatement(sqlOne);

        pstmt1.executeUpdate();

                Savepoint sp1 = con.setSavepoint("sp1");
```

Managing Database Transaction

```
String sqlTwo ="insert into student values "+"(102 ,'ram',97)";

        pstmt1  = con.prepareStatement(sqlTwo);

       pstmt1.executeUpdate();

       Savepoint sp2 = con.setSavepoint("sp2");
```

Managing Database Transaction

```
String sqlThree ="insert into student values "+"(103 ,'vikcy',77)";

 pstmt1  = con.prepareStatement(sqlThree);


pstmt1.executeUpdate();

        con.rollback(sp1);

        con.commit();
} catch(SQLException e) {

        e.printStackTrace();
}
```