IT23039

(1) Demonstrate how a child class can access protected member of its parents class in the same package. Explain with example what happens when the child class is in a different package.

In java a protected member can be accessed
1. within the same package (like default access)
2. In a subclass even if the subclass is in a different package.

Case-1: Child class in the same package

parent class (Same package)

```
Package Pack1;
public class parent {
    protected int value = 10;
```

IT23039

```
    protected void show() {
        System.out.println("value: " + value);
    }
}
```

Child class: (Same package)

```
package pack1;
public class child extends parent {
    public void display() {
        System.out.println(value);
        show();
    }
    public static void main(String[] args) {
        child c = new child();
        c.display();
    }
}
```

- Child class can directly access protected variable and method.
- Work just like package-level access.
- No inheritance restriction issues.

IT23039

Child class (Different package)

```
Package pack2;
import pack1.parent;
Publie class child extends parent{
  publie void display(){
    System.out.println(value);
  }Publie static void main (String[] args){
      child c = new child();
  }   c.display();
}
```

- protected member is accessible through inherita
- Access via this.value or directly is allowed.

IT23039

2. Compare abstract classes and interfaces in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface.

Ans:

Abstract class:

- A class can extend only one abstract class.
- Does not support multiple inheritance of classes.

```
abstract class A {
    abstract void show();
}
abstract class B {
    abstract void display();
}
class C extends A, B { }
```

I123039

Interface

- A class can implement multiple interfaces.
- supports multiple inheritance.

```
interface A {
        void show();
}
interface B {
        void display();
}
class c implements A, B {
        public void show() {
        System.out.println("Show");
        }
public void display() {
        System.out.println("Display");
}
}
```

IT23039

Difference Between Abstract class and interface.

Feature	Abstract class	Interface
Multiple inheritance	Not supported	Supported
Methods	Abstract + Concrete	Abstract methods
Variables	Instance variable allowed	only public static final constants
Constructors	Yes	NO
Access modifiers	Any	Methods are public by default
State	can have state	No instance state

Use an abstract class when:

(i) we want share common code.

(ii) we need to maintain state.

(iii) classes are closely related.

(iv) we want protected / private members.

(v) We don't me need multiple inheritance.

Example:

```
abstract class vehicle{
    int speed;
    abstract void move();
    void setspeed (int s) {
        speed = s;
    }}
```

Use an interface when:

(i) we need multiple inheritance

(ii) we want to define a contract / capability

(iii) classes are unrelated but share behavior.

(iv) we want loose coupling

(v) we want to support future extensions.

```
interface Flyable {
    void fly ();
}
interface Swimmable {
    void swim();
}
```

IT23039

```
class Duck implements Flyable, Swimmable {

        public void fly() {
            system.out.println ("Duck flies");
    }
    public void swim() {
            system.out.println("Duck swim");

    }
}
```

How

IT23039

(3) How does encapsulation ensure data security and integrity? Show with a BankAccount class using private variables and validated methods such as SetAccountNumber(string) SetInitialBalance(double) that rejects null, negative or empty values.

Ans:

Encapsulation means wrapping data (variables) and method together and restricting direct access to data using access modifiers like private.

How it ensures data security

- Data members are declared private.
- Outside classes cannot modify data directly.
- Access is controlled only through methods.

How it ensures data integrity:

- Setter methods validate input.
- Invalid data (null, empty, negative values) is rejected.
- Object always stay in a valid state.

BankAccount Example :

```
class BankAccount {

    private String accountNumber;

    private double balance;

public void setAccountNumber (String accountNumber)
   {
     if (accountNumber == NULL || accountNumber.trim().isEmpty
                                                    ()) {

      System.out.println("Invalid account number!");

     return ;
   }
    this.accountNumber = accountNumber;
}
public void SetInitialBalance (double balance) {

      if (balance < 0) {
       System.out.println ("Initial balance cannot be
                                    negative!");

    return;
}
 this.balance = balance.
}
```

```
public string getAccountNumber() {
    return accountNumber;
}
  public double getBalance() {
      return blo balance;
  }
}
```

Test class

```
public class main {
  public static void main(String[] args){
    BankAccount account = new BankAccount();
account.SetAccountNumber(" "); //rejected
account.setAccountNumber(null); //rejected
account.setAccountNumber("Acc1 2345");
account.setInitialBalance(-500); //rejected
account.setInitialBalance(1000);
System.out.println("Account Number: " + account.
                    getAccountNumber());
System.out.println("Balance : " + account.getBalance
                    ());
```

```
    }
}
```

Output :

```
Invalid account number!
Invalid account number!
Initial balance cannot be negative!
Account Number : ACC12345
Balance : 1000.0
```

I123039

5. Developing a multithreading - based project to Simulate a car parking management system with classes namely - RegistrearParking - Represents a parking request made by a car; parkingpool - Acts as a Shared synchronized queue; parking Agent - Represented a thread that Continously checks the pool and parks cars forom the queue and a Mainclass - Simulates N cars arriving concurrently to request parking.

Car ABC123 requested parking.

Car XYZ456 requested parking

Agent 1 parked car ABC123.

Agent 2 parked car XYZ456

...

Class Responsibilities

1. RegistrearParking (parking request)

- Represents a car requesting parking.
- Holds car number.

2. ParkingPool (Shared Resource)

- A synchronized queue.
- Cars are added by main thread.
- Cars are removed by ParkingAgent threads.

3. ParkingAgent (Consumer Thread)

- Continously checks the pool.
- parcks cars one by one.
- Multiple agents work concurrently.

4. Mainclass

- Simulates N car arriving concurrently.
- starts multiple parking agents.

1. RegistrarParking.java

```
Class RegistrarParking {
    private String CarNumber;
    public RegistrarParking (String carNumber) {
        this.carNumber = CarNumber;
    }
    public String getcarNumber() {
        return CarNumber;
    }}
```

2. parkingPool.java

```
import java.util.LinkedList;
import java.util.Queue;

class parkingPool {
    private Queue<RegistrarParking> queue = new
                    LinkedList<>();
    public synchronized void addCar (RegistrarParking
                                        car) {
      queue.add (car);
    System.out.println("car "+car.getCarNumber()+"requested
                                        Parking.");
     notifyAll(); //notify waiting agents
    }
    public synchronized RegistrarParking getCar(){
        while (queue.isEmpty()){
            try{ wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        return queue.poll();
    }}
```

3. parkingAgent.java

```
class parkingAgent extends Thread {
    private parkingPool pool;
    private int agentID;
public parkingAgent (parkingpool pool, int agentId){
        this.pool = pool;
        this.agentId = agentId;
}
@ Override
    public void run() {
      while (true) {
      Registrarparking car = pool.getcar();
System.out.println("Agent " + agentId + "parked car" +
                                car.getcarNumber() + ".");
try { Thread.sleep (1000); // simulate parking
    } catch (InterruptedException e){
        e.printStackTrace ();
    }}
}
```

4. Mainclass.java

```
Public class Mainclass {
  public static void main (String[] args){
    Parking pool = new ParkingPool ();
  // start parking agents.
      ParkingAgent agent1 = new ParkingAgent(pool,1);
      ParkingAgent agent2 = new ParkingAgent (pool, 2);
    agent1.Start ();
    agent2.start();
String[] cars ={"ABC123","XY2456","LMN789","DEF456"
                "GHI999"};
for ( String carNumber : cars) {
   RegistrarParking car =new RegistrarParking(carNumber);
     pool.addCar (car);
try {
     Thread.Sleep (500); // car arrival gap
   }
     Catch (InterruptedExeeption e) {
     e.printStackTrace ();
```

```
    }
  }
}
}
```

Output :

```
Car ABC123 requested parking.
Car XYZ456 requested parking.
Agent 1 parked car ABC123.
Agent 2 parked car XYZ456.
Car LMN789 requested parking
Agent 1 parked car LMN789
car DEF456 requested parking
Agent 2 parked car DEF456
...
```

12. Describe how JDBC manages communication between a Java program application and a relational database. Outline the steps involved in executing a SELECT query and fetching results. Include error handling with try-catch and finally blocks.

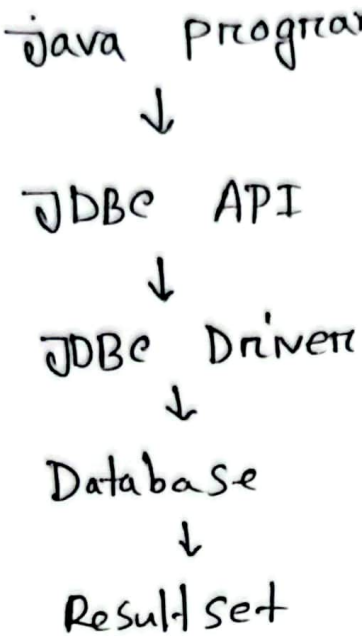
How JDBC Manages Communication

JDBC (Java Database Connectivity) is an API that allows a Java application to connect, send SQL commands and receive results form a relational database.

JDBC works as a bridge between:

- Java Application
- JDBC API
- JDBC Driver
- Relational Database (MySQL / Oracle / PostgreSQL)

Communication Flow :



The JDBC driver translates java method calls into database-specific SQL commands and returns the results back to java.

Steps to Execute a SELECT Query in JDBC

Step 1: Load JDBC Driver

Registers the driver with the DriverManager.

Step 2: Establish Connection

Connects java application to the database.

Step 3: Create Statement

Used to send SQL queries.

. Step 4: Execute SELECT Query

Executes SQL and returns a Resultset.

Step 5: process Resultset

Fetches rows column by column

Step 6: close Resources

prevents memory leaks.

JDBC SELECT Example with Error Handling

```
import java.sql.*;
public class jbs.JdbcSelectExample {
    public static void main (String[] args) {
        Connection con = null;
        Statement stmt = null;
        Resultset rs = null;
    try {
        // Step: 1 Load JDBC Driven
    Class.forName ("com.mysql.cj.jdbc.Driver");
    // step 2 : Establish connection
```

```
Con = DriverManager.getConnection (
        "jdbc:MySQL://localhost:3306/studentdb",
        "root",
        "password"
    );
Stmt = con.createStatement();
String SQL = "SELECT id, name, marks from
                    students";
rs = stmt.executeQuery(sql);
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    double marks = rs.getDouble("marks");
    System.out.println(id + " " + name + " " + marks);
}
catch (ClassNotFoundException e) {
    System.out.println("JDBC Driver not found!");
    e.printStackTrace();
}
catch (SQLException e) {
System.out.println("Database error occurred!");
    e.printStackTrace();
}
```

```
finally {
    try { if (rs! =null) rs.close();
          if (stmt!= null) stmt.close();
          if (con! = null) con.close();
        } catch (SQLException e) {
              e.printStackTrace();
    }
  }
}
```

try block
- Contains databases operation
- Risk of SQL or driver error

Catch block
- Handles:
- ClassNotfoundExpression → driver missing
- SQLException → Connection/query error.

finally block:
- always executes
- Ensures database resources are properly closed

Advantages of JDBC

(i) platform independent

(ii) Source secure database access

(iii) Support multiple databases

(iv) Handle large result sets

(v) Built-in exception handling:

(17) In a Java EE application, how does a servelet Controller manage the flow between the model and the view? provide a brief example the demonstrates forwarding data form a servlet to a JSP and rendering a response.

In a Java EE application:

Model:
- Contains business logic and data.
- Example: Java classes, database access.

Controller (servlet)
- Receives HTTP request
- Interacts with the model
- Places data into request
- Forwards the request to a JSP

View (JSP)
- Retrieves data from request
- Renders HTML response to the client

Request flow

Browser Request

↓

Servlet (Controller)

↓

Model (Business Logic / Data)

↓

Servlet sets attributes

↓

JSP (view)

↓

HTML Response

1. Servlet Controller (Controller)

```
import java.io.IOExeception;
import javax.servlet.*;
import javax.servlet.http.*;
public class StudentServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletExeception, IOException{
```

```
String studentName = "Afrin";
int marks = 85;
request.setAttribute("name", studentName);
request.setAttribute("marks", marks);
RequestDispatcher rd = request.getRequestDispatcher
                       ("student.jsp");
rd.forward(request, response);
}
}
```

2. JSP View (Student.jsp)

```
<html>
<head>
<title> student Result</title>
</head>
<body>
<h2> Student Details </h2>

<P> Name: ${name} </P>
<P> Mark: ${marks} </P>
```

```
<c:if test="${marks>=40}">
  <p> status: pass </p>
  </c:if>
  </body>
</html>
```

[22] How does preparedStatement improve performance and security over Statement in JDBC? Write a short example to insert a record into a MYSQL table using preparedStatement.

PreparedStatement Improve performance.

- SQL query is precompiled by the database.
- Some query can be executed multiple times with different values.
- Reduces parsing and compilation overhead.
- Faster than statement for repeated execution.

PreparedStatement Improve Security.

- Uses parameterized queries.
- Prevents SQL Injection attacks.
- User input is treated as data, not SQL code.

Statement (unsafe)

```
String sql = "INSERT INTO user VALUES ('"+name+"','"+pass+"');
```

preparedStatement (safe)

```
String sql = "INSERT INTO user VALUES (?,?)";
```

Difference Between Statement and PreparedStatement.

Feature	Statement	preparedstatement
SQL injection protection	No	Yes
Precompiled query	No	Yes
Performance	Slower	faster
Parameter Support	No	Yes
Reusability	Low	High

M Insert Record using PreparedStatement (JDBC)

MySQL Table

```
CRETE TABLE Students (
    id INT,
    Name VARCHAR (50),
    marcks DOUBLE
);
```

```
import java.sql.*;
Public class preparedstatementExample {
  Public static void main (String[] args) {
        Connection con = nl null;
        preparedstatement ps = null;
      try {  class.forName ("com.mysql.cj.jdbc.Driver");
          Con =DriverManager.getconnection(
              "jdbc:mysql:// localhost : 3306/studentdb"
               "root"
               "password"
            );
        String sql=INSERT INTO students(id,name,marks)VALUES
                                          (?,?,?)";
        ps = con.preparedStatement (sql);
        ps.setInt (1, 101);
        ps.setString (2, "Afrin");
        ps.setDouble (3, 88.5);
        ps.executeupdate ();
System.out.println ("Record inserted Successfully!");
} .
```

```
catch (SQLException e) {
    System.out.println("Database errrore!");
    e.printStackTrace();
} finally {
    try { if (ps != null) ps.close();
          if (con != null) con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

true if a row exists othe

23. What is Resultset in JDBC and how is it used to retrieve data form a SQL MySQL database? Briefly explain the use of next(), getString(), and getINt() methods with an example.

Resultset in JDBC : A Resultset is an object in JDBC that holds the data returned by executing a SELECT query.

It works like a cursor that points to one row of data at a time.

- Initially, the cursor is positioned before the first row.
- We use next() to move row by row.
- Data is retrieved column-wise using getter methods.

Resultset is used:

next()

- Moves cursor to next row
- Result Return true if a row exists otherwise flase.

Important Resultset Methods

next()

- Move cursor to the next row.
- Returns true is a row exists, otherwise false.

getString(columnName / columnIndex)

- Retrieves String data from the current row

getInt(columnName / columnIndex)

- Retrieves integer data from the current row.

Example: Retrieving Data from MySQL

MySQL Table (students)

id	name	marks
1	Afrin	85
2	Mili	80

```
import java.sql.*;
Public class ResultsetExample {
   public Static void main (String [] args) {
      try {
          Class.forName ("com.mysql.cj.jdbc.Driver");
      Connection con = DriverManager.getConnection(
       "jdbc:mysql://localhost:3306/Studentdb",
        "root",
        "password".
      );
  Statement Stmt = con.createStatement();
  ResultSet rs = Stmt.executeQuery(
      "SELECT id, name, marks. FROM Students"
   );
  While (rs.next()) {
       int id = rs.getInt ("id");
       String name = rs.getString ("name");
       int marks = rs.getInt ("marks");
System.out.println (id + " " + name + " " + marks);
```

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

Output:

```
1   Afrin   85
2   Mili    80
```

26 Design a simple CRUD application using spring Boot and MySQL to manage student record, Describe how each operation (Create, Read, Update, Delete) would be implemented using a repository interface.

A CRUD application perform four basic operations.

Create – Insert new data

Read – Retrieve existing data

Update – Modify existing data

Delete – Remove data

In Spring Boot, CRUD operations are easily implemented using Spring Data JPA and a reposit interface, withe reduce boilen plate JDBe code.

Overall Structure

Client (Browser / postman)

↓

Controller (REST API)

↓

Service Layer (optional but recommanded)

↓

Repository Interface (Jpa repository)

↓

MySQL Database

2. Student Entity (Model

```
import jakarta.persistence.*;
@ Entity
@ Table (name = "student").
Public class Student{
 @ Id
@ Generated Value ( strategpe = Generationtype.IDENTI
     private long id;
     private String Name;
     private int marks;
     // getters and setters
}
```

(i) Map directly to student table in MySQL

(ii) @Id +@ GeneratedValue handles primary key

3. Repository Interface

```
import org.springframework.data.jpa.repository.
        JpaRepository;
public interface StudentRepository
        extends JpaRepository<Student, Long>{

}
```

(i) No method implementation needed

(ii) Spring Data JPA automatically provides CRUD method.

4. CRUD Operation using Repository

CREATE - Insert a student

```
StudentRepository.save(Student);
```

(i) if id is null, JPA perform INSERT

(ii) Used for creating new student records.

READ - Fetch student data

Read · all students

List <Student> students = studentRepository.findAll();

Read by ID

Student student = StudentRepository.findById(id).

orElse(null);

findAll() → SELECT *

findById() → SELECT · where id = ?

Update - Modify Student Record

Student, student = StudentRepository.findById(id),

orElse(int);

Student.setMarks(90);

StudentRepository.save(Student);

(i) if id exit → JPA perform update

(ii) Same save() method handles both create and update.

(iii) DELETE - Remove student

StudentRepository.deleteById(id);

(i) Delete record using primary key

(ii) Internally executes DELETE query.

5. Controller example

```
@ Restcontroller
@ Request mapping ("/students")
public class student Controller {
    @ Autowired
    private student Repository repository;
    @ post mapping
    Public Student create(@ requestBody student s){
    return repository.save(s);
    }
@ getMapping
    public List< student > getAll () {
    return Repository findAll();
    }
@ putMapping ("/{id}")
public student update @ path variable Lang id,
                @ Request Body student s) {
```

```
    s.setId(id);
    return repository.save(s);
}
@DeleteMapping("/{id}")
public void delete(@PathVariable Long id){
    repository.deleteById(id);
}}
```

6. Application properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/ed...
spring.datastructure.username = root
spring.datasource.password =
spring.data.jpa.hibernate.ddl=auto=update
spring.jpa.show-sql = true.
```

Operation	Repository method
Create	Save()
Read (All)	findAll()
Read (ById)	findById()
Update	Save()
Delete	deleteById()

27. How does Spring Boot simplify the development of RESTful Service? Describe how to implement a REST Controller using @RestController, @GetMapping and @PostMapping, including JSON data handling.

Spring Boot reduces complexity by providing:

Auto-Configuration

- Automatically configures web server
- NO XML Configuration needed

Embedded server

- Built-in tomcat → no external server setup

Starter Dependencies

- spring-boot-starter-web includes:
- Spring MVC
- REST support
- Jackson (JSON)

Annotation-Based Development

- Simple annotations like @RestController, @GetMapping, @postMapping

REST Controller Basics

Annotation	Purpose
@RestController	Marks class as REST controller
@GetMapping	Handles HTTP GET Requests
@postMapping	Handles HTTP POST requests
@RequestBody	Converts JSON → java object
@ResponseBody	Converts java object → JSON

Example : Simple REST API with JSON

1. Model class (Student.java)

```
public class Student {
    private int id;
    private String name;
    private double marcks;
    public Student() { }
```

```
public Student (int id, String name, double marks){
        this.id = id;
        this.name = name;
        this.marks = marks;
}
  Public int getId() {return id;}
  Public void setId (int id) {this.id = id;}
  public String getName() { return name;}
  Public void setName (String name) {this.name=name;
  Public double getMarks () { return marks;}
  public void setMarks (double marks){ this.mark
                                    = marks;}
}
```

2. REST Controller (StudentController.java)

```
import org.springframework.wed.bind.annotation.*;
import java.util.*;
@RestController
@RequestMapping ("/students")
```

```
public class StudentController {

    private List<Student> student = new ArrayList<>();

@ GetMapping
  public List<Student> getAllStudents() {
    return students;
  }
@ PostMapping
  public Student addstudent(@ RequestBody Student
                                  Student) {
      Students.add (Student);
      return student;
  }
}
```

JSON handling work.

POST Request (JSON → Java)

```
{"id": 1,
 "name": "Afrin",
  "mark": 88.5 }
```

- @ RequestBody uses Jackson
- Converts JSON into student Object automatically

GET Response (Java → JSON)

```
{
    "id": 1,
    "Name": "Afrin",
    "mark": 88.5
}
```

- Spring Boot automatically converts objects to JSON
- No manual Serialization needed.

32 Demonstrate the project you developed with the important codes and Graphical User Interface.

The project demonstrates how a Spring Boot Java application communicates with a relational database using JDBC and displays the fetched data in a graphical web interface.

Backend: Spring Boot + JDBC

fronted (GUI): HTML + thymeleaf

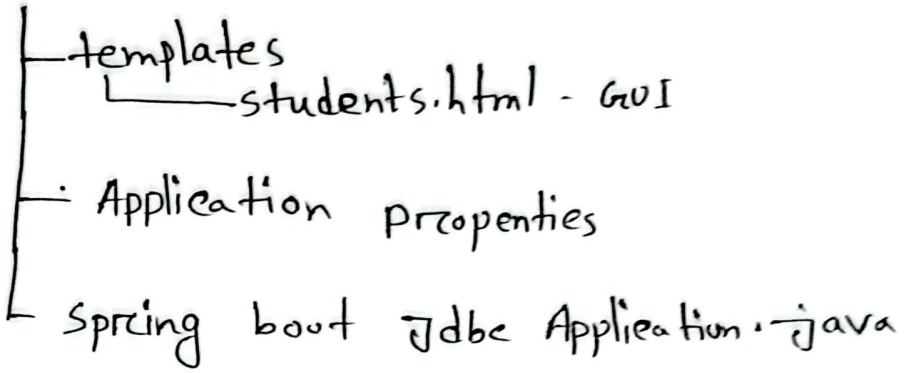
Database: MySQL

Operation: Execute SELECT query and show result in a table.

Update Project Structure

Spring boot - Jdbc - project

- controller
  - student Controller.java
- Model
  - student.java
- Repository. student Repository.java



Student Repository.java

```
@ Repository
public class Student Repository {
    public List<student> get All Students() {
     List<student> list = new ArrayList<>();
   try {
   Connection con = Driver Manager.get Connection(
      " jdbc :mysql://localhost:3306/student db",
       " root ",
       " password"
      );
   Statement stmt = con.create statement();
   result set rs = stmt.execute Qurery (
   " SELECT id, name, cgpa From students "
   ) {
```

(*) GUI displays record in table.

```
while (rs.next()){
        list.add(new Student(
                rs.getInt("id");
                rs.getString("name");
                rs.getDouble("cgpa");
            );
    } catch (SQLException e){
            e.printStackTrace();
    } return list;
  }
}
```

2. Controller connection Backend to GUI Student

Controller.java

```
@ Controller
  public class student Controller{
  private final Student Repository reposity;
  Public Student Controller(Student Repository repository)
      { this.repository = repository;
  }
```

```
@GetMapping ("/students")
    public String viewStudents (Model model){
        model.addAttribute ("Students",
            repository.getAll students ());
        return "student";
}
}
```

3. Graphical User Interface

student.html

```
<! DOCTYPE html>
<html xmlns:th = "http://www.thymeleaf.org">
<head>
    <tittle> Student List </tittle>
    <style>
    body { font-family : Arial;
    background-colour : #f4f6f8;
}
table { bonder-collapse : collapse;
    width : 607;
    mangin : 50px auto;
    background : white;
}
```

```
th, td {
    padding : 12px;
    border : 1px solid #ccc;
    text-align: center;
}
th {
    background-colour: #2c3e50;
    colour: white;
}
h2 { text-align : center;
    margin-top : 40px;
}
</style>
</head>
<body>
<h2> Student Information (Spring Boot JDBC </h2>
<table>
<tr>
<th> ID </th>
</tr>
<tr th:each = "s: ${students}">
    <td th:text = "${s.id}" ></td>
    <td th:text = "${s.name}" ></td>
```

```
<td th:text = "${s.cGpA">
</tr>
</table>
</body>
</html>
```

4. GUI output

http://localhost:8080/students

(i) Clean web page
(ii) Student data displayed in table format
(iii) Data fetch directly form database.

Example :

ID	Name	CGPA
Afi 01	Afrin	3.82
02	Mili	3.80

5. How the whole system works

(i) user opens /students in browser
(ii) Controller receives request
(iii) Repository execute JDBC SELECT query
(iv) Data fetched using ResultSet
(v) Data set to thymeleat page
(vi) GUI displays record in table.