1. Demonstrate How a child class can access a protected member of its parent class within the same package.

Accessing protected member in the same package

Parent.Java

```
package.pack1;
public class parent{
 protected String message=" Hello from P",
```

child.Java

```
package pack1;
 public class ~~void~~ child extends parent{
   public void showmessages(){
    system.out.println(" child accessed "+ messages);
   }
  public state void main (String [] args){
        child c= new child();
        c.showmessage();
   }
  }
```

Since both classes in the same package the protected member message is accessible in child class.

protected in different classes

```
Package pack1;
Public class parent{
 protected string message = "Hello";}
package pack2;
import pack1.parent;
public class child extends parent{
Public void showmessage()
 {
  System.out.println(" Access from child: "+message);
 }
public static void main(String[] args){
    chid c = new chi ld();
    c.showmessage();}}
```

A sub class in a different package can access protected member of the parent class only through inheritance, not ~~enoug~~ through the object of the parent.

2. Compare abstract classes and interfaces in terms of multiple inheritance.

Feature	Abstract class	Interface
Multiple inheritance	Not supported.	Fully supported.
extends	class A extends class B	class A implents x,y,
code reuse	Can have method and member variable.	Java 8+ can have default and static methods.
State	Can have instance variables	only constants (public static final)
constructor	Yes (can Initialize field)	No constructor allowed

When to use interface:

① you want to define pure behaviour, not implementation.

② you want to use multiple inheritance of type.

③ classes are unrelated but share common capabilities.

3. How does encapsulation ensure data security and integrity ? show with an bank account class using private variables and validated methods such as setAccountNumber(String).

⇒ Encapsulation is a key principle in object-oriented programming that hide internal

```
Public void setAccountNumber (String AccountNumber)
{
 if (accountNumber == null || accountNumber trim() isempt
{throw new IllegalArgumentException("can't be null");
}
 this. accountNumber = accountNumber ; }
public void setInitialBalance (double balance){
  if (balance <0){
  throw new IllegalArgument Exception(" Balance can't
                                    be negative"); }
     this. balance = balance;
  }
public String getAccountNumber (){
    return accountNumber;}
   Public double getBalance (){
      return balance;}
   public void deposit (double amount){
     if (amoun>0) this. balance += account;
   }
}
```

4. i) Find kth Smallest Element

```
import java.util.*;
Public class kth Smallest{
Public static void main (String [] args){
List<Integer> list = Arrays.asList (8,2,5,1,9,4);
Collection.sort(list);
int k=3;
System.out.println(k+"rd Smallest"+list.get(k-1));
}
}
```

output = 4.

(ii) Word Frequency using TreeMap.

```
import java.util.*;
Public class wordfreq{
Public static void main (String [] args){
String text= "apple banana apple mango";
TreeMap <String, Integer>map= new TreeMap<>();
for (String word: tex.split (" "))
map.put (word, map.getOrDefault (word,0)+1);
map.forEach((k,v) → system.out.print(k+"="+v));}}
```

Output: apple = 2
banana = 1
mango = 1

(iii) queue and stack using priority queue.

```
import java.util.*;
public class PQstackQueue{
    static class Element{
    int val = order;
Element(int v, int o){ val=v; order = o;}}
public static void main (String [] args){
PriorityQueue <Element> stack = new PriorityQueue<>
((a,b)->b.order->a.order);
priorityQueue <Element> queue= new priorityQueue<>
(comparator.comparing-Int (a->a.oriden));
    int order=0;
 Stack.add (new Element.(10, order++);
 Stack.add (new Element(20, order++);
System.out.println("stack pop: "+stack.poll().val)
 queue.add (new Element(100,0));
 queue.add (New Element (200,1));
system.out.println("queue poll:"+queue.poll().v
}}
```

5. Multithread based project

```
import java.util.*;
import java.util.concurrent.*;
class parkingpool{
private final queue<String> queue=new LinkedList<>(
public synchronized void addcar(String car){
    queue.add(car);
  System.out.println("car"+car+" requested
                        parking.");
  notify();}
 public synchronized String getcar(){
   while(queue.isEmpty()){
   try{wait();}catch(InterruptedExceptione
  {}
   return queue.poll();
    }
  }
```

```
class RegisterParking extends Thread{
  private final string carNumber;
   private final parkingpool pool;
  public RegisterParking (string carNumber, parking
                          pool pool){
  this.carNumber= car Number;
   this.pool= pool}
  public void run(){
     pool.addcar (carNumber);}}
class ParkingAgent extends Thread{
   private final parkingpool pool;
   private final int agentId;
  public ParkingAgent (ParkingPool pool, int agentId
  { this.pool= pool;
    this.agentId= agentId;}
    public void run(){
     while (true){
      String car= pool.getcar();
System.out.println(" Agent" +agentId +" parked
                        car +car+". ");
try{Thread.sleep(500);} Catch( Interrupted Exception
}}}
```

```
public class CarParkingSystem{
public static void main (String[] args) {
 ParkingPool pool = new ParkingPool();
new ParkingAgent (pool,1).start();
new ParkingAgent (pool,2).start();
new RegisterParking ("ABC123", pool).start();
new RegisterParking ("XYZ456", pool).start();
}
}
```

Output:

Car ABC123 requested parking

Car XYZ456 requested parking

Agent 1 parked car ~~XYZ~~ ABC123

Agent 2 parked car XYZ456

6. comparison between DOM vs SAX.

Feature	DOM	SAX
Memory	High (Loads whole xml)	Low (Reads line by lin
speed	slower for big files	faster for large file
Navigation	Easy (Tree structure)	Hard
Modification	Yes	No
Best for	small xml, editing	Large xml

7. How does the virtual DOM in React improve performance?

⇒ React creates a virtual COP of DOM. on updat React.

① compares (diffs) old and new virtual DOM.

② Finds whats changed.

③ Applies only the changes to real DOM.

8. Event delegation in JavaScript.

A Technique where a single event listener is attatched to a parent element to handle events from current and future child elements.

```
<ul id="menu">
  <li> Home </li>
  <li> About </li>
</ul>
document.getElementById("menu").addEventListener(
    "click", function(e){
    if (e.target.tagName === "LI"){
    alert ("clicked :" te.tagret.innertext);
    }
  }
);

// dynamically added item
const newItem = document.createElement('li');
newItem.textcontext = "Item3";
```

document.getElementById("list").appendChild(viewItem);

</script>.

Listener is added only on <ul>, not each <li>.

Works for new <li> like "Item 3" too.

DOM traversal via e.target handles delegation.

9 Explain how Java Regular expression can be used for input validation.

⇒ In Java, Regular Expression (regEx) are used to validate, search or extract patterns from strings such as validating emails, passwords, phone-numbers etc.

Java provides 2 core classes for regex.

- Java.util.regex.Pattern: compiles the regex pattern.
- Java.util.regex.Matcher: Applies the pattern to input string.

Email validation Regex:

String regex = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9+_.-]+$"

This pattern checks:

username: letter, digits, +, -, ., -.

@ symbol

Domain name - letters, digits, dots, hyphen.

```
import java.util.regex.*;
public class EmailValidation{
  public static void main(String[] args){
  String email = "test.user@gmail.com"
  String regex = "^[A-Za-z0-9+_.-]+@
        [A-Za-z0-9.-]+$";
  Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(email);
if (matcher.matches()){
  System.out.println("vaid email");
  else{
    System.out.println("Invalid email");
  }
}
}
```

Pattern.compile() → compiles the regex

matcher() → applies the patten to the input string.

Q. custom annotations are user defined metadata that can be added to code elements (classes, methods etc) and processed at runtime using reflection.

steps to use custom Annotations with Reflection.

① Define the annotation using @ interface
② Annotate elements with it.
③ Use reflection to read and process the annotation at runtime.

Let's build a custom annotation @RunImmediately that makes method to be invoked automatically at runtime.

Define the annotation:

```
import java.lang.annotation.*;
@Retention(RetentionPoliey.RunTime) // Keeps annotation info during Runtime
@Target(ElementType.Method) // can be applied to methods only.
public @Interface RunImmediately {
    int times() default 1; }
```

times(): optional attribute to control how many times to run the method.

Use Annotation in a class:

```
Public class MyService{
@RunImmediately (times= 3)
 Public void sayHello(){
  System.out.println ("Hello");
 }
```

Use Reflection to process Annotation at Runtime:

```
import Java.lang.reflect Method;
 public class AnnotationProcessor{
 public Static void main (String[]args)throws exc-{
 MyService Service = new MyService();
Method [] methods = MyService.class.get
                    Declared Methods();
for (Method method : methods){
if (method is AnnotationPresent (Run ... .class)
 { Run... annotation= method.getAnno(
 for (int i=0; i<annotation.times(); i++){
  method.invoke (service);
}}
}}
```

// The singleton pattern ensures that only one instance of a class exists and provides a global access point to it.

It solves-

① presents multiple instances.

② controls resource usage and ensures consistency.

Basic singleton implementation:

```
public class singleton {
  private static singleton instance;
  private singleton(){ }
  public static singleton getInstance(){
    if (instance == null){
      instance = new singleton(); }
    return instance;
  }
}
```

Thread-safe singleton:

```
public class singleton {
  private static volatile singleton instance;
```

```
private singleton(){}
public static singleton getInstance(){
    if (instance ==null){
     synchronized (singleton.class){
       if (instance ==null){
          instance= new singleton();
       }
     }
    }
    return instance;
  }
}
```

LAB 4: 12 JDBC is an API that enables Java applications to connect and interact with relational databases (like MySQL, PostgreSQL, oracle etc). JDBC manages communication.

① Driver Manager loads the appropriate database driver.

② Establishes a connection to the database.

③ sends SQL queries to the DB engine.

④ Retrives Resultset for Select queries..

⑤ closes resources (Resultset, Statement, connecti after use.

```
import Java.sql.*;
Public class JDBCExample{
 public static void main (String args[]){
    Connection conn = null;
    Statement stmt = null;
    Resultset rs = null;
   try{
class.forName ("com.mysql.cj.Jdbc.Driver");
// Load Driver
conn = DriverManager.getConnection("JDBD:
 mysql://localhost:3306/testdb", "root","pass";
```

```
// connection to DB.
 Stmt = conn.createStatement(); // create statement
rs = stmt.executeQuery("SELECT * from users");
// execute SELECT query
while (rs.next()){
System.out.println(rs.getString("username");
} // process result.
}
 catch (exception e){
e.printStackTrace();}
  finally {
     try{
          if (rs != null) rs.close();
          if (stmt != null) st.close();
          if (conn != null) conn.close();
     }
     catch (SqlException ex){
          ex.printStackTrace();
 }
 }
}
```

13. In a web application following the Mvc pattern:

- servlet acts as the controller: It handles client requests, processes input, interacts with the model and decides which view to display.
- Java class acts as the model: It contains the business logic and data (like fetching data from the database or processing it).
- JSP acts as the view: It presents data to the user in the form of HTML pages. displaying the results sent by the controller.

They work together:

① The client sends a request to the servlet.
② The servlet calls the model class to process the data or fetch required information.
③ The model returns data to the servlet.
④ The servlet sets this data as request attributes and forward the request to a JSP page (view)
⑤ The JSP uses the data to display the output to the user model (Java class).

```
Public class user{
  private String name;
  public User (String name){ this.name= name;}
  public string getname(){ return name;}
}
```

controller (Servlet):

```
protected void doGet(HttpServlet Request request,
      HttpServlet Response response) throws Servlet
        Exception, IOException{
      User user= new User("choity");
      request.setAttribute("User", user);
RequestDispatcher rd = request.getRequestDispatcher
  rd.forward(request, response, ("welcome.jsp");
}
```

View (JSP - welcome.JSP);

```
<html>
<body>
    <h1> welcome, {user name} </h1>
 <body>
<html>.
```

14. The life cycle of a Java servlet is managed by the servlet container. It involves three main stages, handleded by three important methods.

1. init () method:

⇒ The init() method is called once when the servlet is first baded into memory.

⇒ It's used to initialize resources, such as database connections or configure settings.

2. Service () method:

⇒ The Service () method is called each time a client makes a request.

⇒ It determines the HTTP request type (Get, post) etc) and cally the appropriate method.

⇒ This is the where the main request handing logic goes.

3. Destroy () method:

⇒ The destroy() method is called once when the servlet is being removed from memory.

⇒ It's used to release resources, such as closing database connection.

Handling Concurrent Requests:

⇒ A servlet is single instance and multithreaded.

⇒ This means the servlet container creates only one instance of the servlet, but can handle multiple requests at the same time using different threads.

Thread safety issues:

Since multiple threads share the same servlet instance, shared variables can lead to race conditions.

example:

```
private int counter=0;
protected void doGet(---){
    Counter++;
}
```

15 In a servlet, the container creates only one instance, and multiple requests are handled using separate threads. If shared resources are accessed or modified by these threads simultaneously, it can lead to thread safety issues like:

- Race conditions
- Incorrect data
- Unpredictable behaviour.

Example:

suppose we have a servlet that counts how many users visited a page.

```
public class counter Servlet extends HttpServlet{
    private int counter=0;
    protected void doGet (HttpServletRequest req,
                          HttpServletResponse res)
    throws ServletException, IOException{
            counter++;
    res.getWriter().println("visitor number"
                            +counter);
    }
}
```

Solution using synchronization:

```
public class counterServlet extends HttpServlet
{ private int counter=0;
protected void doGet(HttpServletRequest req,
                     HttpServletResponse res)
throws ServletException, IOException{
    Synchronized (this) {
      counter++;
res.getWriter().println("Visitor number",
                        +counter);
  }
 }
}
```

The synchronized block ensures that only one thread can access the counter at a time.

This prevents race conditions and ensure data consistency.

16 The Model-view-controller pattern is a design architecture that separates concerns in a Java web Application. It divides the application into 3 components.

(i) Model (M):

⇒ Represents the data and business logic.

⇒ contains Java classes.

⇒ Handles database interactions, validations and computations.

(ii) view(V):

⇒ Represents the presentation layer.

⇒ Implemented using JSP, HTML, CSS.

⇒ Displays data received from the controller but doesn't handle logic.

(iii) controller(c):

⇒ Acts as the intermediary between Model a view.

⇒ Implemented using servlets.

⇒ Handles requests, processes input, updates mo selects the appropriate view.

Example: Student Registration system.

- View (JSP): Regist JSP - displays a registration form to the user.
- controller (Servlet): RegisterServlet. Java - receives the form data, calls the model and forwards to use a success/ failure page.
- model (Java class): Student. Java, Student DAO. Java - handles business logic and stores Student data into the database.

Advantages:

① Separation of concerns:

Each component has a distinct role, making the codebase clean and organized.

② Maintainability:

changes in UI don't affects business logic and vice versa

③ Reusability:

same model logic can be reused in different views.

(iv) Scalability:

Easier to scale and add new features.

(v) Testability:

Logic is isolated in the model, which makes unit testing easier.

Lab 5:

17 How servlet controller Manages Flow (MVC)

In a Java EE web app, the servlet acts as a controller.

It:

- Takes user input (request).
- Uses the model (Java class) to process data.
- Then forwards data to JSP (view) to show results.

Sample Example:

① Model (User.Java)

```
public class User{
  private string name, email;
Public User( String name, String email){
  this. name= name;
  this. email = email; }
Public String getName(){ return name;}
Public String getEmail(){ return email;}
}
```

(ii) Controller (UserController.java)

```
@WebServlet("/user")
Public class UserController extends HttpServlet{
  Protected void doPost(HttpServletRequest req,
                        HttpServletResponse res)
throws ServletException, IOException{
    String name= req.getParameter("name");
    String email = req.getParameter("email");
  User user = new User(name, email);
  req.setAttribute("User", user);
RequestDispatcher rd = req.getRequestDispatcher
                       ("userInfo.JSP");
  rd.forward(req, res);
  }
}
```

(iii) View (UserInfo.jsp)

⇒
```
<%@page import="model.user"%>
<%
  User user = (User) request.getAttribute("user");
%>
<h2> User Info </h2>
<p> Name: <%= user.getName() %> </p>
<p> Email: <%= user.getEmail() %> </p>
```

18 comparison between cookies, URL Rewritting, HTTP session.

Aspeek	Cookies	URL Rewritting	HTTP session
1. Where data is Stored	On client browser	In the URL as query parameters	On the Server.
2. visibility to user	Yes	Yes	No
3. Security level	Low	Low	High
4. Data limit	Limited (4KB per cookie)	Limited (due to length URL)	Large (depend on server memory.
5. Implementation complexity	Medium	High	Low.

Advantages, Limitations and Ideal use cases;

Cookies:

Advantages: Can persists data even after browser is closed and good for user preference.

Limitations: can be disabled by user. Low security - data is visible and modifiable.

Ideal use cases: Remembering username or preference tracking Returning users

URL Rewritting:

Advantages: Works when cookies are disabled, Easy to implement for small apps or links.

Limitations: Exposes data in URL (security risk) Must Rewrite every URL manually.

Ideal use cases: Temporary session tracking when cookies are turned off. Basic apps without login.

HttpSession:

advantages: Secured and easy to use with built-in support in Java. Automatically expires after timeout.

Limitations: Uses server memory. Needs cleanup to avoid memory leaks in large apps.

Ideal use cases: Login system, shopping carts, personal dash boards.

19 In a web application, HttpSession is used to store user specific data, such as login information, accross multiple requests, when a user logs in, the server creates a session object using request.getSession() and stores attributes like username or userID in that session.

Each user is assigned a unique sessionID, which is typically stored in a cookie on the user's browser. This ID is sent automatically with each request, allowing the server to recognize the user and retrive their session data.

⇒ Working accross Multiple Requests:
When a user logs in, the server creates a session and gives a unique sessionID. This Session ID is sent with every request. The server uses the ID to find the session and show user specific data.

⇒ Session Timeout and Invalidation:

If the user is inactive, the session ends automatically. The session can also be ended using session.invalidate(). After timeout or logout, the user must login again.

⇒ Security Handling:

User data is kept on the server, not on the browser, SessionID should be sent over HTTPS to avoid hacking.

20

(a) controller:

Marks the class as a controller that handles web requests.

(b) RequestMapping:

Maps a specific URL (like/login) to a method inside a controller.

model:

A container used to pass data from the controller to the view.

In spring Mvc, when a browser sends an HTTP request, the framework follows the model-view-controller (Mvc) design pattern to handle and respond to it in a structured way.

Request Handling Flow:

(i) The request first goes to the Dispatcher servlet (the front controller)

(ii) It looks for the correct method to handle the request using annotation like @RequestMapping

(iii) The method is inside a class maked with @controller, which acts as the controller in MVC.

(iv) Business logic is executed and necessary data is added to a Model object.

(v) Finally the view is selected and returned with the model data for display.

Example: (Login form submission)

1. User submits a form to /login.
2. Springs calls the controller method.

```
@controller
Public class LoginController{
@RequestMapping ("/login")
Public String login (@RequestParam String
                     username, model model){

if (username.equals ("admin")){
model.addAttribute ("message", login successfull)
}
else{
model.Add Attribute ("message", Login Failed);}
return "login";
}
}
```

2] In Spring MVC, the DispatcherServlet acts as the front controller, which means it's the central point for receiving all HTTP requests in a web Application.

Role of DispatcherServlet in request Workflow:

(i) Receives Request: The DispatcherServlet receives the request from the browser.

(ii) Finds Handler: It consults handle mapping to find the correct @controller method based on the URL.

(iii) calls controller: It calls the mapped method in the controller to execute business logic.

(iv) Gets view Name: The controller method returns a logical view name.

(v) Resolves view: The dispatcherServlet uses a view resolver to map the logical name to an actual view file.

(vi) Renders view: The view is rendered with the data from the model, and the final HTML is sent back to the browser.

How Dispatchenservlet interacts with handle Mappings and view resolves:

Handler Mapping: Dispatcher Servlet uses it to find the matching @ controller method based on the request URL.

View Resolver: After the controller returns a view name, Dispatchenservlet uses the view resolver to map that name to an actual view file.

Lab-6

22

Performance: Prepared Statement precompiles the SQL query once and reuse it with different parameters. This makes it faster than statement, especially for repeated operations.

Security: It prevents SQL injection by safely binding user input as parameters instead of inserting raw strings into the query.

Example: (Insert Record using PreparedStatement)

```
import Java.sql.*;
Public class insertExample{
Public static void main (String[] args){
  try{
Connection conn = DriverManager.getConnection
("JDBC:mysql://localhost:3306/testdb", root,
"password");
String query = " INSERT INTO Students (name,
        email) values (?,?)";
```

```
PreparedStatement pst= con.prepareStatement(query)
    pst.setString(1, "Alice");
    pst.setString(2, "alice@example.com");
    int nows = pst.executeUpdate();
    System.out.println(nows + " record inserted")
      con.close();}
      catch (Exception e){
        e.printStackTrace();
      }
    }
}
```

```
PreparedStatement pst= con.prepareStatement(query)
    pst.setString(1, "Alice");
    pst.setString(2, "alice@example.com");
    int nows = pst.excuteupdate();
    system.out.println(nows + " record insented")
      con.elose();}
      eateh (Exception e){
       e.printstaekTrace();
    }
  }
}
```

Lab-7

23 In JDBC, a Resultset is an object that hol the result of a SQL SELECT query. It acts like a table in memory and allows you to retrive and process row one by one.

To retrieve data:

(i) A SQL select query is executed using statement or Prepared Statement.

(ii) The result is stored in a Resultset objec

(iii) The next() method is used to move throu each row.

(iv) Methods like getInt(), getString() etc are used to read column values from the current row.

Example:

```
Resultset rs = stmt.executeQuery("SELEC id,
                name from students");

while (rs.next()){
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println("ID:" +id+" Name "+name);
}
}
```

24 JPA (Java Persistence API) is a standard for mapping Java objects to relational database tables. It simplifies database ~~tables~~ operations by allowing developers to work with Java object instead of writing raw SQL queries.

Mapping works:

(i) @Entity : Marks a Java class as a database entity

(ii) @Id : Marks a field as the primary key.

(iii) @GeneratedValue: Automatically generates the primary key value.

JPA manages the mapping between Java objects and relational database tables using annotations. Each Java class is treated as Entity.

Example:

```
import Jakarta.Persistence.*;
@entity
 Public class Student {
  @Id
  @GeneratedValue (strategy = GenerationType.
                                IDENTITY)
   Private Long id;
   private String name;
   private String email; }
```

Here, the student class will be mapped to a table named student. The id field is the primary key, and its value will be automatically generated. The name and email fields will be mapped to columns in the table.

Advantages of JPA over JDBC:

① **Less code:** JPA reduces boilerplate (no need to write SQL for basic opera

② **Object Oriented:** Works directly with objects instead of rows and columns.

③ **Automatic Mapping:** Maps classes to tables w annotations like @entity.

④ **Built in caching:** Improves performance automatic caching.

⑤ **Database Independent:** Easier to switch databases without changing much code.

25 Difference between Entity manager's persists(), merge(), remove().

Persist(objectEntity): Adds a new entity instance to persistence context. Makes the entity managed and schedules for insert in the database when transaction commits. use when creating and saving a new object for the 1st time.

merge (object Entity): updates an existing entity by copying the state of the given detached entity into the merged entity. Returns managed entity. Use when you have a detached object and want to update the database with its changes.

remove(object entity): Marks a managed entity for deletion from the database upon transaction commit. use when you want to delete an existing entity.

Lab 8

26 Goal: Manage student records with fields like id, name, email and course.

Technologies used:

- Java
- Springboot
- Spring Data JPA
- MySQL

1) Entity class: Student.java

```
import. Jakarta.persistence.*;
@entity
Public class student{
  @Id
  @Generatedvalue (Strategy= GenerationType.
                                    IDEN
    private Long id;
    private String name;
    private String email;
    Private String course;
}
```

2 Repository Interface : StudentRepository. Java

```
import org. spring framework. data. Jpa. repository
. JPA repository
public interface Student Repository extends Jpa-
                repository <student, long>
        { }
```

3 Service Layer (Student service. Java)

```
import org. spring framework. beans. factory. annotation.
                                        autowired;
import org. spring framework. stereotype. service;
import Java. util. List;
import Java. util. optional;
@ service
public class Studentservice{
@ autowired
  private StudentRepository repository;
// create
public student saveStudent (Student student){
    return repository. save (student); }
//read all
public List<student> getAllStudents(){
    return repository. findAll (); }
```

```
public Optional<Student> getStudentById(Long id)
    {return repository.findById(id);}

Public Student updateStudent(Long id, Student updatedData){
Student student = repository.findById(id).orElseThrow();
 Student.setName(updated data.getName());
Student.setcourse(updated data.getcourse());
return repository.save(student);}

public void deleteStudent(Long id){
  repository.deleteById(id);
}}
}
```

CRUD operation using Repository:

Create:

```
Student student = new student();
Student. setName ("Alice");
Student. setemail ("alice@example.com");
StudentRepositony. save (student);
```

Read

Get all students: List<Student> students = stude
Repositony findAll(

GetbyID: Optional<Student> student= student
Repositony. find by ID (i

update.

Retrieve entity, update fields and save student

```
existing. student Repositony. find by ID (id).
existing. set email (" new email @ example. com
StudentRepositony. save (existing)
```

Delete!

```
StudentRepositony delete By Id (id);
```

Lab 9

27 The process how spring Boot simplifies RESTful Service Development.

- Auto configuration: spring Boot auto-configures RESTAPI using built-in support for JSON, TomCAT server etc.
- Reduced Boilerplate: No need to write XML or complex on complex setup.
- Embedded Server: comes with an embedded server (Like Tomcat) to run immediately.
- Built in JSON support: Automatically converts Java objects to JSON and vice versa using Jackson.

@

① Restcontrollen.

creating a Model class (Student. Java)

```
Public class student{
    Private string name;
    private string course;
}
```

(11) Create a REST controller (StudentController.java)

```
@RestController
@RequestMapping("/student")
Public class studentcontroller{
  List<student> StudentList = new ArrayList<>();
  @GetMapping
    Public List<student> getstudents(){
        return studentlist;}

  @postMapping
Public String addstudent(@RequestBody student
                                    student){

    StudentList.add(student);
    return "student added successfully!";

  }
}
```

JSON Example:

{"name" : "choity"
"course" : "ICT"}.