Lab-1 :

Multiple Inheritance

Abstract class:

- Java does not ~~do~~ support multiple inheritance with classes.
- A class can extend only one abstract class.

Interface:

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

When to Use Abstract class:

- Classes are closely related
- You want to share common code

Use interface when -

- You need multiple inheritance
- You want to define capability/contract.

Lab 2:-

How encapsulation ensures security:-

1. Data is hidden using private.
2. Access allowed only via validated methods.
3. Prevents direct modification of critical data.

BankAccount Example:

```
class BankAccount {
    private String accountNumber;
    private double balance;
    public void setAccountNumber(String accNo){
        if(accNo == null || accNo.trim().isEmpty()){
            throw new IllegalArgumentException("Invalid
            account number");
        }
        this.accountNumber = accNo;
    }
    public void setInitialBalance(double amount){
        if(amount < 0){
            throw new IllegalArgumentException("Balance
            cannot be negative");
        }
        this.balance = amount;
    }
    public double getBalance(){
        return balance;
    }
}
```

Lab-3:

Project Overview

- RegistrarParking - represents a parking request
- ParkingPool - shared synchronized queue
- ParkingAgent - Worker threads that park cars
- MainClass - simulates multiple cars arriving concurrently

Cars = producers

Agents = consumers

1. RegistrarParking (ParkingRequest):

```
public class RegistrarParking {
    private String carNumber;
    public RegistrarParking (String carNumber) {
        this.carNumber = carNumber;
        system.out.println ("car" + carName + "requested
          parking.");
    }
    public string getCarNumber () {
        return carNumber;
    }
}
```

2 ParkingPool (Shared Synchronized Queue):

```
import java.util.*;
public class ParkingPool{
    private Queue<RegistranParking> queue = new LinkedList<>();

    public synchronized void addCar(RegistranParking car){
        queue.add(car);
        notifyAll();
    }
    public synchronized RegistranParking getCar(){
        while(queue.isEmpty()){
            try{
                wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
        return queue.poll();
    }
}
```

3. Parking Agent :

```
public class ParkingAgent extends Thread {
    private ParkingPool pool;
    private String agentName;
    public ParkingAgent(ParkingPool pool, String agentName){
        this.pool = pool;
        this.agentName = agentName;
    }

@Override
public static void run(){
    while(true){
        RegistrarParking car = pool.getCar();
        System.out.println(agentName + "parked car"
        + car.getCarNumber());

        try{
            Thread.sleep(1000);
        }catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}
```

4. MainClass

```
public class MainClass{
    public static void main(String[] args){
        ParkingPool pool = new ParkingPool();
        ParkingAgent agent1 = new ParkingAgent(pool,"Agent 1");
        ParkingAgent agent2 = new ParkingAgent(pool," Agent 2");
        agent1.start();
        agent2.start();
        String [] cars ={" AB123", "XYZ456", " LMN789",
                DEF 321", "JKL654" };
        for(String carNo : cars){
            new Thread() ->{
                RegistrarParking car = new RegistrarParking
                (carNo);
                pool.addCar(car);
            }).start();
            try {
                Thread.sleep(500);
                } catch(InterruptedException e){
                    e.printStackTrace();
                }
            }
        }
    }
```

Lab-9 :

How JDBC Works

1. Load Driver
2. Create connection
3. Create Statement
4. Execute query
5. Process ResultSet
6. close resources

Example :

```
try {
    Connection con = DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/students", "root", "");

    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("Select * from student");

    while(rs.next()){
      System.out.println(rs.getString("name"));
      }
    }catch (Exception e){
        e.printStackTrace();

    }finally{
        System.out.println("Query Executed");

    }
```

Lab-5

In Java EE applications, the servlet acts as a controller in the MVC (Model-View-Controller) architecture. It receives client requests, interacts with the model and forwarded data to the view (JSP).
The servlet sets attributes in the request object and forwards the request to a JSP page using a RequestDispatcher. The JSP then renders the response dynamically.

Flow:
servlet (controller) → Model → JSP (view)

Servlet:

```
@WebServlet("/student")
public class StudentServlet extends HttpServlet {
        protected void doGet(HttpServletRequest req, HttpServletRes
        ponse res)
        throws ServletException, IOException {
        req.setAttribute("name", "Bhuban");
        RequestDispatcher rd = req.getRequestDispatcher
         ("student.jsp");
         rd.forward(req, res);
        }
}
```

JSP (student.jsp)

```
<h2>Welcome ${name}</h2>
```

Lab-6:

PreparedStatement improves performance and security compared to Statement. It prevents SQL injection attacks by separating SQL Logic from data values. Prepared statements are pre compiled and reused making execution faster. PreparedStatement is preferred when executing the same SQL query multiple times with different values.

Example:

```
String sql = "Insert into student(name,age) values(?,?)";
PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1, "Bhuban");
ps.setInt(2,21);
ps.executeUpdate();
```

Lab-7:

ResultSet is an interface used to store data retrived from a database after executing a SELECT query. It maintains a cursor that points to the current row of data.

.next() moves the cursor to the next row

.getString() retrives string data

.getInt() retrives integer data

These methods allow efficient traversal and retrival of database records.

Example:

```
ResultSet rs = st.executeQuery("SELECT* from student");
while(rs.next()){
    String name = rs.getString("name");
    int age = rs.getInt("age");
    System.out.println(name+" "+age);
}
```

Lab-9:

Spring Boot simplifies RESTful service development by providing embedded servers, auto configuration, and seamless JSON support.

Using @RestController, REST API's can be created easily.

@GetMapping handles HTTP GET requests, while @PostMapping handles POST requests.

Data is exchanged in JSON format using @RequestBody.

This approach enables efficient communication between client and server applications.

REST controller example:

```
@RestController
@RequestMapping("/students")
public class StudentController{

    @GetMapping
    public List<Students> getAll(){
        return repo.findAll();
    }

  @PostMapping
    public student add(@RequestBody Student s){
        return repo.save(s);
    }

}
```