2) Compare abstract and interfaces.

Key Differences —

Abstract Class:

→ Extends one class only

→ Has abstract + concrete methods

→ Can have fields, constructors

→ methods can be any access level

Interface:

→ Implements multiple interfaces

→ methods are abstract by default

→ Only constants, no constructors

→ Methods are public only

When to use —

Use abstract class when

→ Classes are closely related

→ You need shared code

→ You need protected private members

Use interface when

-> Need multiple inheritance

-> Unrelated classes need same behaviour

-> Only contract needed.

Example

Scenario: Animal kingdom

Abstract class approach:

-> Animal (abstract class) with method eat()

-> Dog extends animal

-> Cat extends animal

Problem: Dog can also extend Robot

Interface approach:

-> Walkable (interface) with method walk()

-> swimmable (interface) with method swim()

-> Duck implements both Walkable and Swimmable

-> Amphibious Robot implements both.

Summary:

Abstract class can be used to just share code among classes. Interface can be used for multiple inheritance, achieve polymorphism etc.

3) Encapsulation : Data Security and Integrity

Encapsulation protects data by using private variables and validated methods.

How it ensures security -

→ Private fields prevent direct access
→ Data can only be modified through class methods
→ Prevents unauthorized or accidental changes

How it ensures integrity -

→ Setter methods validate input
→ Invalid values (null, empty, negatives) are rejected
→ Object always stays in a valid state

Example BankAccount Class:

```
class BankAccount {
    private string accountNumber;
    private double balance;
    public void setAccountNumber (String accNo) {
        if(accNo != null && !accNo.isEmpty())
            accountNumber = accNo;
    }
```

```
    public void setInitialBalance(double amount) {
        if (amount >= 0)
            balance = amount;
    }
    public String getAccountNumber() {return accountNumber;}
    public double getBalance() {return balance;}
}
```

Key Points -

-> Direct access like account.accountNumber = null is not allowed

-> Validation enforces business rules

-> Invalid data never enters the object.

Conclusion -

Encapsulation ensures data security by hiding variables and data integrity by validating all updates through methods.

5) Multithreaded Car Parking Management System.

Aim

To implement a multithreaded car parking with synchronization.

Description

Multiple car request parking concurrently. Parking agents (threads) safely process requests from a synchronized queue.

Class

→ Registrar Parking → ParkingAgent

→ ParkingPool → MainClass

Algorithm

1. Create synchronized parking pool
2. Add parking requests
3. Start parking agents threads
4. Agents process requests safely

Java Example Parking Management System Program

class RegistrarParking { String carNo; RegistrarParking(String c) { carNo = c; }

```
class ParkingPool {
    java.util.Queue<RegistrarParking> q = new java.util.LinkedList<>();
    synchronized void add(RegistrarParking r){ q.add(r); notify(); }
    synchronized RegistrarParking get() throws Exception {
        while (q.isEmpty()) wait();
        return q.poll();
    }
}
class ParkingAgent extends Thread {
    ParkingPool p; int id;
    ParkingAgent(ParkingPool p, int i) { this.p = p; id = i; }
    public void run() {
        try {
            System.out.println("Agent" + id + "parked" + p.get().carNo);
        } catch(Exception e) {}
    }
}
public class MainClass {
    public static void main(String[] a) {
        ParkingPool p = new ParkingPool();
        p.add(new RegistrarParking("ABC123"));
        p.add(new RegistrarParking("XYZ456"));
        new ParkingAgent(p,1).start();
        new ParkingAgent(p,2).start();
    }
}
```

Sample Output

Car ABC123 requested parking.

Car XY2456 requested parking.

Agent 1 parked car ABC123.

Agent 2 parked car XY2456.

Conclusion

The system demonstrates thread-safe handling of concurrent parking requests using synchronization.

Result

Lab objective achieved.

13) ☐ JDBC Communication with Database

Aim

To understand JDBC communication and execute a SELECT query with proper exception handling.

Description

JDBC (Java Database Connectivity) allows Java applications to communicate with relational databases using database drivers. It provides a platform-independent way to execute SQL queries and retrieve results.

Steps to execute SELECT Query

1. Import JDBC packages
2. Load JDBC Driver
3. Establish connection
4. Create Statement
5. Execute SELECT query.
6. Process ResultSet
7. Close resources

Java Code (Shiv4):

```
import java.sql.*;
public class JDBCSelect {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/testdb", "user", "password");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT id, name FROM student")) {

        while (rs.next())
            System.out.println(rs.getInt(1)+" "+rs.getString(2));
        } catch (Exception e) {
        System.out.println("Error " + e.getMessage());
    }
}
```

Conclusion :

JDBC enables secure efficient communication.

Result :

SELECT query executed successfully.

17) Servlet Controller in Java EE (MVC)

Aim

To show how a servlet controls flow between model and view.

Role

Model : Data/Logic

View (JSP): Output

Controller (Servlet): Handles request, forward data.

Flow

Request → Servlet → setAttribute → JSP → Response

Code (Main):

Servlet :

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class S extends HttpServlet {
    protected void doGet(HttpServletRequest r, HttpServletResponse s)
    throws Exception {
        r.setAttribute("n", "John");
        r.getRequestDispatcher("v.jsp").forward(r,s);
    }
}
```

JSP : ${n}

Conclusion: Servlet controls request flow and sends model data to JSP.

22) PreparedStatement improvement analysis -

Improvement -

Performance -

-> Pre-compiled SQL

-> Reuse with different values

Security -

-> Uses ? Parameters

-> Prevents SQL injection

Example Code -

```
import java.sql.*;
class 1 {
  public static void main(String[] a) throws Exception {
    Connection c=DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/testdb","root","password");
    PreparedStatement p = c.prepareStatement(
      "insert into student values(?,?,?)");
    p.setInt(1,1); p.setString(2,"Rahmat"); p.setInt(3,21);
    p.execute(); c.close();
  }
}
```

Conclusion - PreparedStatement is faster, safer and preferred over statement

23) Resultset in JDBC and its Uses

Definition:

Resultset stores data returned by a SELECT query and allows row-by-row access.

Methods:

next() → Moves to next row

getString() → reads string column

getInt() → reads int column

Example Code:

```
import java.sql.*;
class F{
  public static void main(String[] a) throws Exception{
    Connection c = DriverManager.getConnection(
      "jdbc:mysql://localhost:3306/testDb", "root", "password");
    Resultset r = c.createStatement()
    .executeQuery("select id, name, age from student");
    while(r.next())
      System.out.println(r.getInt(1)+" "+r.getString(2)+" "+r.getInt(3));
    c.close();
  }
}
```

Conclusion: Resultset retrieves and processes database records using cursor methods.