Program 1:

1. **A) Develop a Java program to demonstrate class fundamentals, including object creation, method implementation, and constructor usage. Utilize features like the this keyword and the finalize() method.**

```
// Demonstrate class fundamentals in Java

class Box {
   double width;
   double height;
   double depth;

   // Constructor using this keyword
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
   }

   // Method to calculate and return volume
   double volume() {
      return width * height * depth;
   }

   // Finalize method called before object is destroyed
   @Override
   protected void finalize() {
System.out.println("Box object is being destroyed.");
   }
}

public class BoxDemo {
   public static void main(String[] args) {
      // Create objects of Box
      Box myBox1 = new Box(10, 20, 15);
      Box myBox2 = new Box(3, 6, 9);

      double vol;

      // Get volume of first box
vol = myBox1.volume();
System.out.println("Volume of myBox1 is: " + vol);
```

```
        // Get volume of second box
    vol = myBox2.volume();
    System.out.println("Volume of myBox2 is: " + vol);

        // Hint for garbage collector (not guaranteed)
        myBox1 = null;
        myBox2 = null;

        // Request garbage collection
    System.gc();  // This may call finalize(), but not guaranteed immediately
       }
    }
```

Output:

Volume of myBox1 is: 3000.0

Volume of myBox2 is: 162.0

Box object is being destroyed.

Box object is being destroyed.

B) Write a java program with a class called box and has three variables width, height and depth assign the value 100 to width, 200 to height, 150 to depth. Compute the volume in the main class.

```
class Box
{
	double height;
	double width;
	double depth;
}

class BoxDemo
{
	public static void main(String args[])
	{
		Box mybox1 = new Box();

		mybox1.height = 100;
		mybox1.width = 200;
		mybox1.depth = 150;
		double volume;

		volume = mybox1.height*mybox1.width*mybox1.depth;
		System.out.println("Volume is: " + volume);
	}
}
```

Output:

Volume is : 3000000.0

C) Create a main class BoxDemo2 create a another class Box with three instance variables where the dimensions are automatically initialized when the object is constructed. The volume must be calculated in Box class and displayed in BoxDemo2 class.

```
class Box
{
	double height;
	double width;
	double depth;
	Box()
	{
		height = 20;
		width = 30;
		depth = 25;
	}
	double vol()
	{
		return height*width*depth;
	}
}
class BoxDemo2
{
	public static void main(String args[])
	{
		Box mybox = new Box();
		System.out.println("Volume is: "+mybox.vol());
	}
}
```

Output:

Volume is :15000.0

Program 2:

Implement a Java program showcasing inheritance concepts, including creating a multilevel hierarchy, using the super keyword, method overriding, and dynamic method dispatch. Also, include the use of abstract classes and final with inheritance.

```
// Abstract base class
abstract class Box {
   double width, height, depth;

   // Constructor to clone object
Box(Box ob) {
       width = ob.width;
       height = ob.height;
       depth = ob.depth;
   }

   // Constructor with dimensions
Box(double w, double h, double d) {
       width = w;
       height = h;
       depth = d;
   }

   // Default constructor
Box() {
       width = height = depth = -1;
   }

   // Constructor for cube
Box(double len) {
```

```
        width = height = depth = len;
    }

    // Abstract method for volume
    abstract double volume();
}

// First level subclass
class BoxWeight extends Box {
    double weight;

    // Clone constructor
BoxWeight(BoxWeightob) {
        super(ob);
        weight = ob.weight;
    }

    // Constructor with all parameters
BoxWeight(double w, double h, double d, double m) {
super(w, h, d);
        weight = m;
    }

    // Default constructor
BoxWeight() {
super();
        weight = -1;
    }

    // Constructor for cube
```

```
BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }

    // Overriding volume method
    double volume() {
        return width * height * depth;
    }
}

// Final class to prevent further inheritance
final class Shipment extends BoxWeight {
    double cost;

    // Clone constructor
Shipment(Shipment ob) {
        super(ob);
        cost = ob.cost;
    }

    // Constructor with all parameters
Shipment(double w, double h, double d, double m, double c) {
super(w, h, d, m);
        cost = c;
    }

    // Default constructor
Shipment() {
super();
```

```
        cost = -1;
    }

    // Constructor for cube
Shipment(double len, double m, double c) {
super(len, m);
        cost = c;
    }

    // Overriding volume method
    double volume() {
        return width * height * depth;
    }

    // Final method - can't be overridden
    final void displayCost() {
System.out.println("Shipping cost: $" + cost);
    }
}

// Demo class
public class DemoShipment {
    public static void main(String args[]) {
        Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.14);

        double vol;
vol = shipment1.volume();
System.out.println("Volume is " + vol);
System.out.println("Weight of the Shipment: " + shipment1.weight);
        shipment1.displayCost();
```

```
System.out.println();

        Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
vol = shipment2.volume();
System.out.println("Volume is " + vol);
System.out.println("Weight of the Shipment: " + shipment2.weight);
        shipment2.displayCost();
    }
}
```

Output:

```
Volume is 3000.0
Weight of the Shipment: 10.0
Shipping cost: $3.14

Volume is 24.0
Weight of the Shipment: 0.76
Shipping cost: $1.28
```

OR

```
// Abstract base class
abstract class Animal {
    String name;

Animal(String name) {
        this.name = name;
    }

    // Abstract method
    abstract void sound();
```

```
    // Final method - cannot be overridden
    final void sleep() {
System.out.println(name + " is sleeping...");
    }
}
// First level subclass
class Mammal extends Animal {
Mammal(String name) {
        super(name); // call superclass constructor
    }
    @Override
    void sound() {
System.out.println(name + " makes a mammal sound.");
    }
}
// Second level subclass (Multilevel Inheritance)
class Dog extends Mammal {
Dog(String name) {
        super(name); // call superclass constructor
    }
    @Override
    void sound() {
System.out.println(name + " barks.");
    }
    void display() {
super.sound(); // call overridden method from superclass
sound();       // call own version of method
    }
}
```

```
// Main class demonstrating dynamic dispatch
public class InheritanceDemo {
    public static void main(String[] args) {
        Animal ref;           // superclass reference

        ref = new Dog("Buddy"); // dynamic method dispatch

ref.sound();          // Dog's sound() will be called
ref.sleep();          // Final method from Animal

        // Casting to access Dog-specific method
        if (ref instanceof Dog) {
            ((Dog) ref).display();
        }
    }
}
```

Output:

```
Buddy barks.
Buddy is sleeping...
Buddy makes a mammal sound.
Buddy barks.
```

Program 3:

Explore the String class and command-line arguments by developing a program that manipulates strings and processes arguments. Additionally, demonstrate the use of varargs and the Scanner class for input.

```
import java.util.Scanner;


class Box {
   double width, height, depth;


Box(double w, double h, double d) {
       width = w;
       height = h;
       depth = d;
   }


   public String toString() {
       return "The dimensions are " + width + " by " + height + " by " + depth + ".";
   }
}


public class ToStringDemo {


   // Method using varargs
   static void printWords(String... words) {
System.out.println("Words passed using varargs:");
       for (String word : words) {
System.out.println(word);
       }
   }
```

```
    public static void main(String args[]) {
        // Box and toString
        Box b = new Box(10, 12, 14);
        String s = "The box b is " + b;
System.out.println(“printing b”+b);    // Converted to string
System.out.println(“printing s”+s);

 // String manipulation
        String s2 = "Face the failure until the failure fails to face you";
int start = 4;
int end = 37;

        char buf[] = new char[end - start];
        s2.getChars(start, end, buf, 0);
System.out.println("Extracted substring using getChars: ");
System.out.println(buf);

        // Command-line arguments
System.out.println("The number of command-line arguments: " + args.length);
        for (inti = 0; i<args.length; i++) {
System.out.println("Argument " + (i + 1) + ": " + args[i]);
        }

        // Varargs method
printWords("Java", "String", "Scanner", "Varargs");

        // Scanner input
        Scanner sc = new Scanner(System.in);
System.out.print("Enter a sentence: ");
        String input = sc.nextLine();
```

```
System.out.println("You entered: " + input);


sc.close();
    }
}
```

Output:

```
javac ToStringDemo.java
F:\MSRIT\JAVA\Lab>java ToStringDemo
The dimensions are 10.0 by 12.0 by 14.0.
The box b is The dimensions are 10.0 by 12.0 by 14.0.
Extracted substring using getChars:
 the failure until the failure fa
The number of command-line arguments: 0
Words passed using varargs:
Java
String
Scanner
Varargs
Enter a sentence: change is constant
You entered: change is constant
```

After giving command line arguments:

```
F:\MSRIT\JAVA\Lab>java ToStringDemo Hello world
The dimensions are 10.0 by 12.0 by 14.0.
The box b is The dimensions are 10.0 by 12.0 by 14.0.
Extracted substring using getChars:
 the failure until the failure fa
The number of command-line arguments: 2
Argument 1: Hello
Argument 2: world
Words passed using varargs:
```

Java

String

Scanner

Varargs

Enter a sentence: this is java lab

You entered: this is java lab

Program 4:

Create a Java program that uses packages and interfaces, demonstrating access protection, importing packages, and implementing interface methods. Include default interface methods in your implementation.

//File: p1/Protection.java

```
package p1;


public class Protection {
int n = 1;
   private intn_pri = 2;
   protected intn_pro = 3;
   public intn_pub = 4;


   public Protection() {
System.out.println("Protection constructor in p1");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
   }
}
```

//File: p1/SamePackage.java

```
package p1;
public class SamePackage {
   public SamePackage() {
      Protection p = new Protection();
System.out.println("SamePackage constructor in p1");
System.out.println("n = " + p.n);
      // System.out.println("n_pri = " + p.n_pri); // Not accessible (private)
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
```

```
    }
}
```

// File: p1/MyInterface.java

```
package p1;

public interface MyInterface {
    void display(); // abstract method

    default void show() {
System.out.println("Default method in MyInterface");
    }
}
```

//File: p1/Demo.java

```
package p1;

public class Demo implements MyInterface {
    public void display() {
System.out.println("Implementation of abstract method from MyInterface");
    }

    public static void main(String[] args) {
        Demo d = new Demo();
d.display();
d.show();

System.out.println("\n--- Access Protection Demonstration ---");
        Protection obj = new Protection();
SamePackagesp = new SamePackage();
```

```
System.out.println("\n--- Calling classes from p2 package ---");
        p2.Protection2 p2Obj = new p2.Protection2();
        p2.OtherPackage op = new p2.OtherPackage();
    }
}
```

//File: p2/Protection2.java

```
package p2;
import p1.Protection;

public class Protection2 extends Protection {
    public Protection2() {
System.out.println("Protection2 constructor in p2 (subclass of Protection)");
        // System.out.println("n = " + n);        // default - not accessible outside package
        // System.out.println("n_pri = " + n_pri); // private - not accessible
System.out.println("n_pro = " + n_pro); // protected - accessible in subclass
System.out.println("n_pub = " + n_pub); // public - accessible
    }
}
```

//File: p2/OtherPackage.java

```
package p2;

import p1.Protection;

public class OtherPackage {
    public OtherPackage() {
        Protection p = new Protection();
System.out.println("OtherPackage constructor in p2");
        // System.out.println("n = " + p.n);      // default - not accessible
        // System.out.println("n_pri = " + p.n_pri); // private - not accessible
```

```
        // System.out.println("n_pro = " + p.n_pro); // protected - not accessible here
System.out.println("n_pub = " + p.n_pub);   // public - accessible
    }
}
```

Output:

```
Implementation of abstract method from MyInterface
Default method in MyInterface

--- Access Protection Demonstration ---
Protection constructor in p1
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
Protection constructor in p1
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
SamePackage constructor in p1
n = 1
n_pro = 3
n_pub = 4

--- Calling classes from p2 package ---
Protection constructor in p1
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
```

Protection2 constructor in p2 (subclass of Protection)

n_pro = 3

n_pub = 4

Protection constructor in p1

n = 1

n_pri = 2

n_pro = 3

n_pub = 4

OtherPackage constructor in p2

n_pub = 4

Program 5:

Develop a Java application to handle various exceptions using try, catch, throw, throws, and finally. Implement nested try statements and create custom exception subclasses.

```
// Custom exception subclass
class MyException extends Exception {
    private int detail;

MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

public class ExceptionHandlingDemo {
```

```
    // Method that throws an exception
    static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
        if (a > 10)
            throw new MyException(a);
System.out.println("Normal exit");
    }

public static void main(String args[]) {
        try {
            // Nested try block
            try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;  // may cause ArithmeticException
intc[] = {1};
c[42] = 99;      // may cause ArrayIndexOutOfBoundsException
            } catch (ArithmeticException e) {
System.out.println("Divide by 0: " + e);
            }

compute(1);    // no exception
compute(20);   // will throw MyException

        } catch (MyException e) {
System.out.println("Caught: " + e);
        } finally {
System.out.println("Inside finally block");
        }
    }
```

```
}
```

Program 6:

Implement a Java program to demonstrate exception handling fundamentals, including catching multiple exceptions and using Java's built-in exceptions.

```
public class MultipleExceptionDemo {
   public static void main(String args[]) {
int a = 10;
int b = 0;
int result;
int[] nums = {1, 2, 3};

      try {
         // Causes ArithmeticException
         result = a / b;
System.out.println("Result: " + result);

         // Causes ArrayIndexOutOfBoundsException
nums[5] = 10;

         // Causes NullPointerException
```

```
            String str = null;
System.out.println(str.length());


        } catch (ArithmeticException e) {
System.out.println("Caught ArithmeticException: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Caught ArrayIndexOutOfBoundsException: " + e);
        } catch (NullPointerException e) {
System.out.println("Caught NullPointerException: " + e);
        }


System.out.println("After try-catch blocks.");
    }
}
```

Program 7:

Create a Java program to illustrate multithreaded programming concepts, including creating and running multiple threads, using isAlive(), join(), thread priorities and Synchronization.

```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {
System.out.print("[" +msg);
    try {
Thread.sleep(1000);
    } catch (InterruptedException e) {
System.out.println("Interrupted");
    }
System.out.println("]");
  }
}
class Caller implements Runnable {
```

```
  String msg;
Callme target;
  Thread t;
  public Caller(Callmetarg, String s) {
    target = targ;
msg = s;
    t = new Thread(this);
t.start();
  }
  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
target.call(msg);
    }
  }
}
class Synch1 {
  public static void main(String args[]) {
Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

System.out.println("ob1 is alive: " + ob1.t.isAlive());
System.out.println("ob2 is alive: " + ob2.t.isAlive());
System.out.println("ob3 is alive: " + ob3.t.isAlive());
    // wait for threads to end
    try {
ob1.t.join();
ob2.t.join();
```

```
ob3.t.join();
   } catch(InterruptedException e) {
System.out.println("Interrupted");
   }
System.out.println("ob1 is alive after join: " + ob1.t.isAlive());
System.out.println("ob2 is alive after join: " + ob2.t.isAlive());
System.out.println("ob3 is alive after join: " + ob3.t.isAlive());
  }
}
```

Program 8:

Develop a Java program that uses enumerations and demonstrates autoboxing with type wrappers.

a)

```
enum Apple {
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

   private int price;

   // Constructor
Apple(int p) {
       price = p;
   }

   // Method
intgetPrice() {
       return price;
   }
}

class EnumClassDemo {
```

```
    public static void main(String args[]) {
        Apple ap = Apple.RedDel;
System.out.println("Price of " + ap + " is " + ap.getPrice());
    }
}
```

b)

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
 public static void main(String args[]) {
            Integer Iob = new Integer(100);//Boxing
            inti = Iob.intValue();//Unboxing
            System.out.println(i + " " + IOb);

            Integer iOb = 100; // autobox an int
            inti = iOb; // auto-unbox
            System.out.println(i + " " + iOb);
             }
}
```

Program 9:

Implement a generic class with multiple type parameters and create a simple example using generics to showcase their usage.

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
 T ob1;
 V ob2;
 // Pass the constructor a reference to
 // an object of type T and an object of type V.
TwoGen(T o1, V o2) {
 ob1 = o1;
```

```
 ob2 = o2;
 }
 // Show types of T and V.
 void showTypes() {
System.out.println("Type of T is " +
 ob1.getClass().getName());
System.out.println("Type of V is " +
 ob2.getClass().getName());
 }
 T getob1() {
 return ob1;
 }
 V getob2() {
 return ob2;
 }
}
// Demonstrate TwoGen.
class SimpGen {
 public static void main(String args[]) {
TwoGen<Integer, String>tgObj =
 new TwoGen<Integer, String>(88, "Generics");
 // Show the types.
tgObj.showTypes();
 // Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
 String str = tgObj.getob2();
System.out.println("value: " + str);
 }
}
```

Program 10:

Develop a Java program to demonstrate the Collections Framework, including the use of ArrayList and LinkedList. Showcase how these collections are used and manipulated.

a) ArrayList

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
 public static void main(String args[]) {
 // Create an array list.
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
 // Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
 // Display the array list.
System.out.println("Contents of al: " + al);
 // Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
```

```
 }
}
```

b) LinkedList

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
 public static void main(String args[]) {
 // Create a linked list.
LinkedList<String>ll = new LinkedList<String>();
 // Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
 // Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
 // Get and set a value.
 String val = ll.get(2);
ll.set(2, val + " Changed");
```

```
System.out.println("ll after change: " + ll);
 }
}
```

Program 11:

Create a Java application utilizing lambda expressions to simplify code and implement block lambda expressions.

a)

```
// Demonstrate a lambda expression that takes two parameters.
interface NumericTest2 {
booleantest(int n, int d);
}
class LambdaDemo3 {
  public static void main(String args[])
  {
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

if(isFactor.test(10, 2))
System.out.println("2 is a factor of 10");

if(!isFactor.test(10, 3))
System.out.println("3 is not a factor of 10");
  }
}
```

Output :

```
 2 is a factor of 10
3 is not a factor of 10
```

b)

```
// A block lambda that computes the factorial of an int value.
```

```
interface NumericFunc {
intfunc(int n);
}
class BlockLambdaDemo {
  public static void main(String args[])
  {
    // This block lambda computes the factorial of an int value.
NumericFunc factorial = (n) ->  {
int result = 1;
for(inti=1; i<= n; i++)
        result = i * result;
      return result;
    };
System.out.println("The factoral of 3 is " + factorial.func(3));
System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

Output :

```
 The factorial of 3 is 6
The factorial of 5 is 120
```

Program 12:

mplement a simple Java program to demonstrate common design patterns such as Singleton, Factory, Observer, and Strategy.

A) Singleton Design Pattern

```
//Save as SingleDP.java
public class SingleDP {
    public static void main(String[] args) {
        Printer p1 = Printer.getInstance();
        p1.print("Hello, World!");
```

```
        Printer p2 = Printer.getInstance();
        p2.print("Java Design Patterns");



        System.out.println(p1 == p2);
    }
}
//Save as Printer.java
// Singleton class
public class Printer {

    private static Printer instance;

    private Printer() {
        System.out.println("Printer is ready.");
    }

    public static Printer getInstance() {
 System.out.println(instance);

        if (instance == null) {
            instance = new Printer();
        }

        return instance;
    }
```

```
    public void print(String message) {
        System.out.println("Printing: " + message);
    }
}
```

B) Factory Design Pattern

```
// Abstract Product Class
abstract class Product {
    public abstract void display();
}

// Concrete Products
class ConcreteProductA extends Product {
    @Override
    public void display() {
        System.out.println("This is Concrete Product A.");
    }
}

class ConcreteProductB extends Product {
    @Override
    public void display() {
        System.out.println("This is Concrete Product B.");
    }
}
```

```
// Creator Abstract Class
abstract class Creator {
    public abstract Product factoryMethod();
}

// Concrete Creators
class ConcreteCreatorA extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}

class ConcreteCreatorB extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}

// Client Code
public class FactoryMethodExample {
    public static void main(String[] args) {
        Creator creatorA = new ConcreteCreatorA();
        Product productA = creatorA.factoryMethod();
        productA.display();
```

```
        Creator creatorB = new ConcreteCreatorB();
        Product productB = creatorB.factoryMethod();
        productB.display();
    }
}
```

C) Observer Design Pattern

```
import java.util.List;
import java.util.ArrayList;

// Observer interface
interface Observer {
    void update(String weatherData);
}

// Concrete Observers
class MobileApp implements Observer {
    public void update(String weatherData) {
        System.out.println("Mobile App: Weather updated - " + weatherData);
    }
}

class Website implements Observer {
    public void update(String weatherData) {
        System.out.println("Website: Weather updated - " + weatherData);
    }
}
```

```
// Subject
class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private String weather;

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void setWeather(String newWeather) {
        this.weather = newWeather;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer o : observers) {
            o.update(weather);
        }
    }
}
public class ObserverDemo {
    public static void main(String[] args) {
```

```
        WeatherStation station = new WeatherStation();

        Observer mobile = new MobileApp();
        Observer site = new Website();

        station.addObserver(mobile);
        station.addObserver(site);

        station.setWeather("Sunny 32°C");
    }
}
```

D) Strategy Design Pattern

```
// Strategy interface
interface TravelStrategy {
    void travel(String destination);
}

// Concrete strategy: Car
class CarTravel implements TravelStrategy {
    public void travel(String destination) {
        System.out.println("Driving to " + destination + " by car. Takes more time but gives flexibility.");
    }
}

// Concrete strategy: Train
class TrainTravel implements TravelStrategy {
    public void travel(String destination) {
```

```
        System.out.println("Going to " + destination + " by train. Comfortable and affordable.");
    }
}

// Concrete strategy: Plane
class PlaneTravel implements TravelStrategy {
    public void travel(String destination) {
        System.out.println("Flying to " + destination + " by plane. Fastest but most expensive.");
    }
}

// Context class
class TravelPlanner {
    private TravelStrategy strategy;

    // Set the travel strategy (can be changed at runtime)
    public void setStrategy(TravelStrategy strategy) {
        this.strategy = strategy;
    }

    // Execute the selected travel strategy
    public void goTo(String destination) {
        if (strategy == null) {
            System.out.println("Please select a travel strategy first!");
        } else {
            strategy.travel(destination);
```

```
        }
    }
}

// Client code
public class StrategyTravelDemo {
    public static void main(String[] args) {
        TravelPlanner planner = new TravelPlanner();

        // Travel using car
        planner.setStrategy(new CarTravel());
        planner.goTo("New York City");

        // Travel using train
        planner.setStrategy(new TrainTravel());
        planner.goTo("New York City");

        // Travel using plane
        planner.setStrategy(new PlaneTravel());
        planner.goTo("New York City");
    }
}
```

Program 13:

Create a Java program to handle various types of events using the Delegation Event Model, including implementing event listener interfaces and adapter classes. Demonstrate event handling mechanisms with key and action events.

A) with key and action events.

```
import java.awt.*;
import java.awt.event.*;

// AWT Frame with KeyListener and ActionListener
public class EventDemo extends Frame implements KeyListener, ActionListener {

    TextField tf;
    TextArea ta;

    public EventDemo() {
        setLayout(new FlowLayout());

        tf = new TextField(20);
        ta = new TextArea(10, 20);
        Button b = new Button("Clear");

        add(tf);
        add(ta);
        add(b);

        // Register listeners
        tf.addKeyListener(this);
        b.addActionListener(this);

        // Window listener to handle closing event
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
```

```
        setSize(300, 300);
        setTitle("Event Demo");
        setVisible(true);
    }

    // KeyListener methods
    public void keyPressed(KeyEvent ke) {
        ta.append("Key Pressed: " + ke.getKeyChar() + "\n");
    }

    public void keyReleased(KeyEvent ke) {
        // Not used in this demo
    }

    public void keyTyped(KeyEvent ke) {
        // Not used in this demo
    }

    // ActionListener method
    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals("Clear")) {
            ta.setText("");
        }
    }

    public static void main(String[] args) {
        new EventDemo();
    }
}
```

B) Adapter Class

```
import java.awt.*;
import java.awt.event.*;

public class AdapterDemo extends Frame {

    public AdapterDemo() {
        setTitle("Adapter Demo");
        setSize(400, 200);
        setLayout(new FlowLayout());

        // Add mouse listeners using adapter classes
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));

        // Close window on click
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        setVisible(true);
    }

    // This replaces showStatus from Applet
    public void showStatus(String msg) {
```

```
        // Set title bar to show the message
        setTitle(msg);
    }

    public static void main(String[] args) {
        new AdapterDemo();
    }
}

class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;

    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }

    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked at (" + me.getX() + ", " +
me.getY() + ")");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;

    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
```

```
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged at (" + me.getX() + ", " +
me.getY() + ")");
    }
}
```

Program 14:

Implement a Java program to connect to a database using JDBC, perform basic CRUD operations, and handle database connectivity.

//Open MySQL Command Line and run:

```
CREATE DATABASE studentdb;

USE studentdb;

CREATE TABLE student (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);

INSERT INTO student VALUES (1, 'John'), (2, 'Alice');
```

//JAVA Program

```
import java.sql.*;


public class JDBCExampleCrud {
    public static void main(String[] args) {
        try {
            // 1. Load the driver class
            Class.forName("com.mysql.cj.jdbc.Driver");


            // 2. Create a connection
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/studentdb", "mcauser1", "msrit@2024");
```

```
        // 3. Create a statement
        Statement stmt = con.createStatement();

        // --- UPDATE operation ---
    String updateSQL = "UPDATE student SET name = 'Ram' WHERE id = 1";
        int rowsUpdated = stmt.executeUpdate(updateSQL);
        System.out.println("Rows updated: " + rowsUpdated);

        // --- DELETE operation ---
        String deleteSQL = "DELETE FROM student WHERE id = 2";
        int rowsDeleted = stmt.executeUpdate(deleteSQL);
        System.out.println("Rows deleted: " + rowsDeleted);

        // 4. Execute query to see updated table data
        ResultSet rs = stmt.executeQuery("SELECT * FROM student");

        // 5. Process the result
        System.out.println("Current student table data:");
        while (rs.next()) {
            System.out.println(rs.getInt(1) + "  " + rs.getString(2));
        }

        // 6. Close the connection
        con.close();
      } catch (Exception e) {
        System.out.println(e);
```

```
        }
    }
}
```