Name: Sariya Mazhar

Enrollment Number: mern02

Batch / Class: Batch-01

Assignment: (Bridge Course Day 6)

Date of Submission:01/07/2025

Problem Solving Activity 1

1. Program Statement: Object Review, Smartphone class

- I want to build a Smartphone class with attributes brand, model, and batteryLevel, and methods turnOn(), turnOff(), and charge(). I should explain how encapsulation protects internal data

---

2. Class: Smartphone

Attributes:

brand (String)

model (String)

batteryLevel (int)

Methods:

turnOn()

turnOff()

charge()

---

3. How encapsulation applies?

- Encapsulation means hiding the internal state of an object and accessing it only through methods (also known as interfaces). In this case:
- The smartphone's attributes (brand, model, batteryLevel) are private.
- You interact with the phone only through public methods like turnOn(), turnOff(), and charge().
- You cannot directly change batteryLevel from outside — it's protected by methods, maintaining the integrity of the data.

4. Program Code

```
class Smartphone {
    private String brand;
    private String model;
    private int batteryLevel;
    public Smartphone(String brand, String model) {
        this.brand = brand;
        this.model = model;
        this.batteryLevel = 50; // default battery level
    }
    public void turnOn() {
        if (batteryLevel > 0) {
            System.out.println(model + " is now ON.");
        } else {
            System.out.println("Battery dead. Please charge first.");
        }
    }
    public void turnOff() {
        System.out.println(model + " is now OFF.");
    }
    public void charge() {
        batteryLevel = 100;
        System.out.println("Charging... Battery is now full.");
    }
    public int getBatteryLevel() {
        return batteryLevel;
    }
}
```

5. Output

```
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %  /u
odeDetailsInExceptionMessages -cp /Users/sariyamazhar/Lib
temUp\ BridgeCourse_994dd6f4/bin Code_1
Galaxy A51 is now ON.
Battery Level: 50%
Charging... Battery is now full.
Battery Level after charging: 100%
Galaxy A51 is now OFF.
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %
```

---

6. Observation / Reflection

Encapsulation keeps batteryLevel private so it cannot be misused.The class provides controlled access through public methods like charge(), turnOn().

Problem Solving Activity 2

1. Program Statement: Conceptual need for Extension

- Design a base class Employee with common attributes and behaviors, and extend it using subclasses Manager and Developer with their own specific attributes and methods.
- Demonstrate how inheritance helps avoid code duplication by reusing shared logic from the Employee class.

---

2. How inheritance helps?

- Inheritance allows subclasses like Manager and Developer to reuse the common code (attributes like name, employeeId, salary and methods like work(), takeBreak()), which is defined once in the Employee class.
- This means we don't need to redefine name, work(), or similar features in every subclass — saving time, reducing redundancy, and improving maintainability.
- Changes in the base class automatically apply to all subclasses — ensuring consistency.

3. Pseudocode

```
Define class Employee
    Declare attributes: name, employeeId, salary
    Define method work()
        Print name is working
    Define method takeBreak()
        Print name is taking a break

Define class Manager inheriting from Employee
    Declare attributes: department, teamSize
    Define method conductMeeting()
        Print name is conducting a meeting for department

Define class Developer inheriting from Employee
    Declare attributes: programmingLanguage, projects
    Define method writeCode()
        Print name is writing code in programmingLanguage

Define main method
    Create object mgr of type Manager
    Set mgr.name to "Alice"
    Set mgr.department to "Sales"
    Call mgr.work()
    Call mgr.conductMeeting()

    Create object dev of type Developer
    Set dev.name to "Bob"
    Set dev.programmingLanguage to "Java"
    Call dev.takeBreak()
    Call dev.writeCode()
```

4. Code:

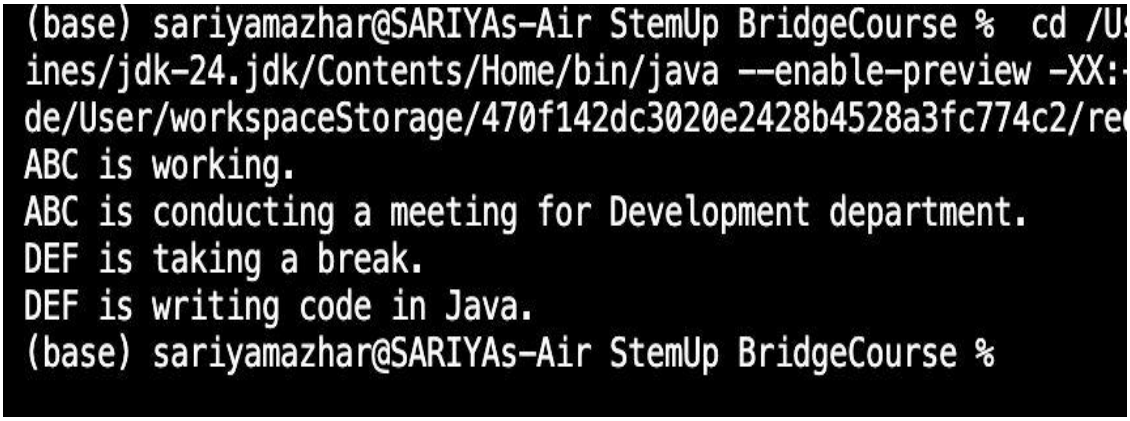
```
class Employee {
    String name;
    int eId;
    double sal;
    void work() {
        System.out.println(name + " is working.");
    }
    void takeBreak() {
        System.out.println(name + " is taking a break.");
    }
}
class Manager extends Employee {
    String department;
    int teamSize;

    void conductMeeting() {
        System.out.println(name + " is conducting a meeting for " + department + " department.");
    }
}
class Developer extends Employee {
    String programmingLanguage;
    int projects;
    void writeCode() {
        System.out.println(name + " is writing code in " + programmingLanguage + ".");
    }
}
public class Code_2 {
    public static void main(String[] args) {
        Manager mgr = new Manager();
        mgr.name = "ABC";
```

```
        mgr.department = "Development";
        mgr.work();
        mgr.conductMeeting();

        Developer dev = new Developer();
        dev.name = "DEF";
        dev.programmingLanguage = "Java";
        dev.takeBreak();
        dev.writeCode();
    }
}
```

5. Output



6. Observation:

- I observed that inheritance helps avoid repetition by letting subclasses inherit and use attributes and methods from the base class, while also allowing customization through additional features.

Problem Solving Activity 3

1. Program Statement: Employee Hierarchy

- To design a base class Employee with attributes name, employeeId, and salary, and a method getDetails() to display employee information.
- Create subclasses Manager and Developer that add specific attributes (department and programmingLanguage) and override getDetails() to include these details.

---

2. Algorithm

Step 1: Define a base class Employee with common attributes and a method to display details.

Step 2: Create a Manager class extending Employee, adding a department attribute.

Step 3: Create a Developer class extending Employee, adding a programming language attribute.

Step 4: Override the getDetails() method in both subclasses to show their specific information.

Step 5: In main(), create objects of Manager and Developer and call their getDetails() method.

---

3. Pseudocode

Class Employee with name, employeeId, and salary

  Constructor initializes attributes

  Method getDetails prints employee info

Class Manager inherits Employee

  Adds department

  Overrides getDetails to include department

Class Developer inherits Employee

  Adds programmingLanguage

  Overrides getDetails to include programmingLanguage

In main method:

  Create Manager object and call getDetails

  Create Developer object and call getDetails

---

4. Code

```
// Base class
class Employee {
    String name;
    int employeeId;
    double salary;
    Employee(String name, int employeeId, double salary) {
        this.name = name;
        this.employeeId = employeeId;
        this.salary = salary;
    }
    void getDetails() {
        System.out.println("Name: " + name);
        System.out.println("Employee ID: " + employeeId);
        System.out.println("Salary: ₹" + salary);
    }
}
class Manager extends Employee {
    String department;
    Manager(String name, int employeeId, double salary, String department) {
        super(name, employeeId, salary);
        this.department = department;
    }
    @Override
    void getDetails() {
        super.getDetails();
        System.out.println("Department: " + department);
    }
}
class Developer extends Employee {
    String programmingLanguage;
```

```
    Developer(String name, int employeeId, double salary, String programmingLanguage) {
        super(name, employeeId, salary);
        this.programmingLanguage = programmingLanguage;
    }
    @Override
    void getDetails() {
        super.getDetails();
        System.out.println("Programming Language: " + programmingLanguage);
    }
}
public class Code_3{
    public static void main(String[] args) {
        Manager m = new Manager("Fatima", 201, 75000, "HR");
        Developer d = new Developer("Zaid", 202, 85000, "Java");

        m.getDetails();
        System.out.println();
        d.getDetails();
    }
}
```

---

5. Output:

```
Name: Fatima
Employee ID: 201
Salary: ₹75000.0
Department: HR

Name: Zaid
Employee ID: 202
Salary: ₹85000.0
Programming Language: Java
```

6. Observation

- This program demonstrates inheritance and method overriding in Java. It uses polymorphism to extend the base class for specialized employee types.

Problem Solving Activity 4

1. Program Statement: Animal Kingdom

- To create a base class: Animal with method makeSound() Subclasses: Dog and Cat, override the method Create and test objects

---

2. Algorithm

Step 1: Define a base class Animal with a method makeSound().

Step 2: Create two subclasses Dog and Cat that override makeSound().

Step 3: In the main method, declare references of type Animal.

Step 4: Assign objects of Animal, Dog, and Cat to the references.

Step 5: Call makeSound() on each reference to observe polymorphic behavior.

---

3. Pseudocode

```
Class Animal:
    Method makeSound():
        Print "The animal makes a sound."
Class Dog inherits Animal:
    Override makeSound():
        Print "The dog barks."
Class Cat inherits Animal:
    Override makeSound():
        Print "The cat meows."
```

Main Method:

Declare myAnimal = new Animal()

Declare myDog = new Dog()

Declare myCat = new Cat()

Call myAnimal.makeSound()

Call myDog.makeSound()

Call myCat.makeSound()

---

4. Code

```
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("The dog barks.");
    }
}
class Cat extends Animal {
    void makeSound() {
        System.out.println("The cat meows.");
    }
}
public class Code_4 {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        myAnimal.makeSound();
        myDog.makeSound();   }}
```

5. Output

```
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %  cd
ines/jdk-24.jdk/Contents/Home/bin/java --enable-preview -X
de/User/workspaceStorage/470f142dc3020e2428b4528a3fc774c2/
The animals makes plenty of sound.
The dog says boww boww.
The cat meowss meows.
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %
```

---

6. Observation

- Polymorphism allows the same method to behave differently based on the object. The makeSound() method is overridden in Dog and Cat to produce unique outputs.

Problem Solving Activity 5

1. Program Statement: Inheritance tree

- Base: Electronic Device Subclasses: Television, Laptop, Smartphone List attributes and methods per subclass

---

2. Algorithm

Step 1: Create a base class ElectronicDevice with methods turnOn() and turnOff().

Step 2: Create subclasses Television, Laptop, and Smartphone that inherit from ElectronicDevice.

Step 3: Add device-specific methods: changeChannel() for Television, runProgram() for Laptop, and makeCall() for Smartphone.

Step 4: In the main method, create objects of Television, Laptop, and Smartphone.

Step 5: Call both inherited and device-specific methods using the objects.

3. Pseudocode:

Class ElectronicDevice

Method turnOn(): Print "Turn the TV ON"

Method turnOff(): Print "Turn the TV OFF"

Class Television inherits ElectronicDevice

Method changeChannel(): Print "Changing TV channel"

Class Laptop inherits ElectronicDevice

Method runProgram(): Print "Running a program on Laptop"

Class Smartphone inherits ElectronicDevice

Method makeCall(): Print "Making a call from Smartphone"

Create tv, call turnOn() and changeChannel()

Create laptop, call turnOn() and runProgram()

Create phone, call turnOn() and makeCall()

---

4. Code:

```
class ElectronicDevice {
    void turnOn() {
        System.out.println("Turn the TV ON");
    }
    void turnOff() {
        System.out.println("Turn the TV OFF");
    }
}
class Television extends ElectronicDevice {
    void changeChannel() {
        System.out.println("Changing TV channel");
    }
}
class Laptop extends ElectronicDevice {
    void runProgram() {
        System.out.println("Running a program on Laptop");
```

```
    }
}
class Smartphone extends ElectronicDevice {
    void makeCall() {
        System.out.println("Making a call from Smartphone");
    }}
public class Code_5 {
    public static void main(String[] args) {
        Television tv = new Television();
        tv.turnOn();
        tv.changeChannel();
        Laptop laptop = new Laptop();
        laptop.turnOn();
        laptop.runProgram();
        Smartphone phone = new Smartphone();
        phone.turnOn();
        phone.makeCall();
    }
```

5. Output

6. Observation

- This program demonstrates inheritance by allowing all device classes to share common functionality.Each subclass also adds its own unique behavior, showcasing polymorphism through method extension.

Problem Solving Activity 6

1. Program Statement:Payment Gateway

- Abstract class: Payment Gateway with abstract processPayment(double amount) Subclasses: Credit CardGateway, PayPalGateway Attempt to instantiate abstract class (should fail)

---

2. Algorithm

Step 1: Start and define an abstract class PaymentGateway with abstract method processPayment(amount).

Step 2: Create subclasses CreditCardGateway and PayPalGateway that override processPayment(amount).

Step 3: In the main() method of class Code6, declare references of type PaymentGateway.

Step 4: Assign objects of CreditCardGateway and PayPalGateway to those references and call processPayment().

Step 5: Note that instantiating PaymentGateway directly will cause a compile-time error.

---

3. Pseudocode

```
Define abstract class PaymentGateway
    Declare abstract method processPayment(amount)
Define class CreditCardGateway inheriting PaymentGateway
    Implement processPayment(amount)
        Print "Credit Card: ₹" + amount
Define class PayPalGateway inheriting PaymentGateway
    Implement processPayment(amount)
        Print "PayPal: ₹" + amount
Define class Code6
    Main method:
        Create object cc of type PaymentGateway assigned to new CreditCardGateway
        Create object pp of type PaymentGateway assigned to new PayPalGateway
        Call cc.processPayment(1000)
        Call pp.processPayment(2500)
```

4. Code:

```
abstract class Payment{
    abstract void processPayment(double amount);
}
class CreditCard extends Payment {
    void processPayment(double amount) {
        System.out.println("My Credit Card has: ₹" + amount);
    }
}
class Phonepe extends Payment{
    void processPayment(double amount) {
        System.out.println("My Phonepe balance is: ₹" + amount);
    }
}
public class Code_6 {
    public static void main(String[] args) {
        Payment a = new CreditCard();
        Payment b = new Phonepe();
        a.processPayment(350000);
        b.processPayment(229000);
    }
}
```

5. Output

```
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %  /u
w -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/sari
4c2/redhat.java/jdt_ws/StemUp\ BridgeCourse_994dd6f4/bin
My Credit Card has: ₹350000.0
My Phonepe balance is: ₹229000.0
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %
```

---

6. Observation

- This program demonstrates abstraction and runtime polymorphism by using an abstract class reference to call overridden methods.It also shows that abstract classes cannot be instantiated directly, enforcing implementation in derived classes.

Problem Solving Activity 7

1. Program Statement: Instrument Sounds

- To create a abstract class: Instrument with abstract play() Subclasses: Guitar, Piano Implement and test

---

2. Algorithm

Step 1: Start the program.

Step 2: Define an abstract class Instrument with an abstract method play().

Step 3: Create subclass Guitar that overrides play() with a specific message.

Step 4: Create another subclass Drum that overrides play() with a different message.

Step 5: In the main() method, create objects of Guitar and Drum using Instrument references.

Step 6: Call the play() method on each object to demonstrate polymorphism.

3. Pseudocode

Class Instrument is abstract

  Abstract Method: play()

Class Guitar extends Instrument

  Method play(): Print "I enjoy playing guitar!"

Class Drum extends Instrument

  Method play(): Print "Playing the drum is all fun!"

Main method:

  Create object g = new Guitar() using Instrument reference

  Create object p = new Drum() using Instrument reference

  Call g.play()

  Call p.play()

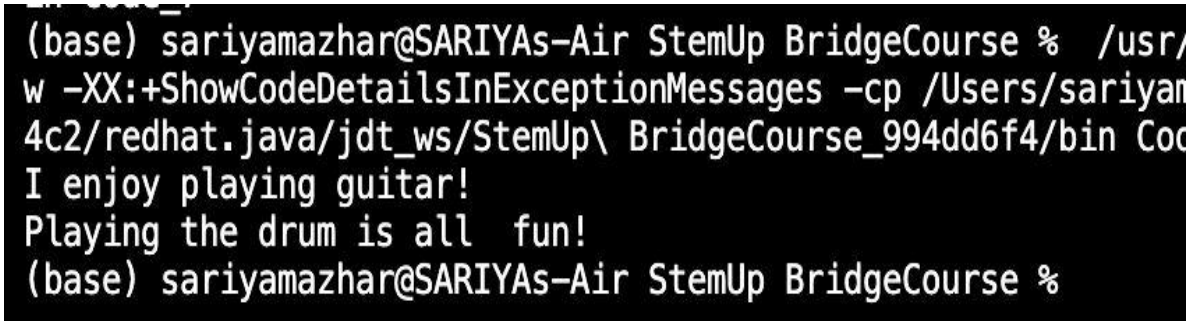
---

4. Code

```
abstract class Instrument {
    abstract void play();
}
class Guitar extends Instrument {
    void play() {
        System.out.println("I enjoy playing guitar!");
    }
}
class Drum extends Instrument {
    void play() {
        System.out.println("Playing the drum is all  fun!");
    }
}
public class Code_7 {
    public static void main(String[] args) {
        Instrument g = new Guitar();
```

```
        Instrument p = new Drum();
        g.play();
        p.play();
    }
}
```

5. Output



6. Observation

- This program demonstrates abstraction and polymorphism by using an abstract class reference to call subclass-specific play() methods.It also reinforces that abstract classes cannot be instantiated directly, ensuring subclass implementation.

Problem Solving Activity 8

1. Program Statement: Abstracting a Task

- To create Automated Task, method execute() Subclasses: EmailSender, FileArchiver, DatabaseBackup Use abstraction to simplify the execution of tasks

---

2. Algorithm

Step 1: Start by creating an abstract class Automation with an abstract method execute().

Step 2: Define subclasses EmailSender, FileArchiver, and DatabaseBackup that override execute().

Step 3: In the main() method, declare references of type Automation.

Step 4: Instantiate the subclasses and assign them to the Automation references.

Step 5: Call the execute() method on each object to perform specific tasks.

---

3. Pseudocode:

```
Class Automation is abstract
    Abstract Method: execute()
Class EmailSender extends Automation
    Method execute(): Print "Sending emails..."
Class FileArchiver extends Automation
    Method execute(): Print "Archiving files..."
Class DatabaseBackup extends Automation
    Method execute(): Print "Backing up database..."
    Create Automation reference for EmailSender
    Create Automation reference for FileArchiver
    Create Automation reference for DatabaseBackup
    Call execute() on each object
```

---

4. Code:

```
abstract class Automation {
   abstract void execute();
}
```

```
class EmailSender extends Automation {
    void execute() {
        System.out.println("Sending emails...");
    }
}
class FileArchiver extends Automation  {
    void execute() {
        System.out.println("Archiving files...");
    }
}
class DatabaseBackup extends Automation {
    void execute() {
        System.out.println("Backing up database...");
    }
}
public class Code_8 {
    public static void main(String[] args) {
        Automation  email = new EmailSender();
        Automation archive = new FileArchiver();
        Automation backup = new DatabaseBackup();
        email.execute();
        archive.execute();
        backup.execute();
    }
}
```

5. Output

```
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %  /usr/bin/
w -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/sariyamazha
4c2/redhat.java/jdt_ws/StemUp\ BridgeCourse_994dd6f4/bin Code_8
Sending emails...
Archiving files...
Backing up database...
(base) sariyamazhar@SARIYAs-Air StemUp BridgeCourse %
```

---

6. Observation

- This program showcases abstraction and polymorphism by defining a common interface execute() in the abstract class Automation. Each subclass provides its specific task implementation, allowing uniform method calls through superclass references.

Problem Solving Activity 9

1. Program Statement: Geometric shapes

- Demonstrating Abstract base: Shape with getArea() Subclasses: Circle, Square Create polymorphic list and calculate areas

---

2. Algorithm

Step 1: Start and define an abstract class Shape with an abstract method getArea().

Step 2: Create a subclass Circle with a radius attribute and implement getArea() using the area formula for a circle.

Step 3: Create another subclass Square with a side attribute and implement getArea() using the square area formula.

Step 4: In the main() method, create an array of Shape objects including both Circle and Square instances.

Step 5: Iterate over the array using an enhanced for-loop.

Step 6: For each shape, print its class name and calculated area using the getArea() method

---

3. Pseudocode

Define abstract class Shape

Abstract Method: getArea()

Class Circle extends Shape

Attribute: radius

Constructor sets radius

Method getArea(): return π × radius × radius

Class Square extends Shape

Attribute: side

Constructor sets side

Method getArea(): return side × side

Main method:

Create array of Shape with Circle and Square objects

For each shape in array

Print shape's class name and its areaEnd function

Read num1

Read num2

4. Code:

```
abstract class Shape {

    abstract double getArea();
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    double getArea() {
        return Math.PI * radius * radius;
    }
}

class Square extends Shape {
    double side;

    Square(double side) {
        this.side = side;
    }

    double getArea() {
        return side * side;
    }
}
```

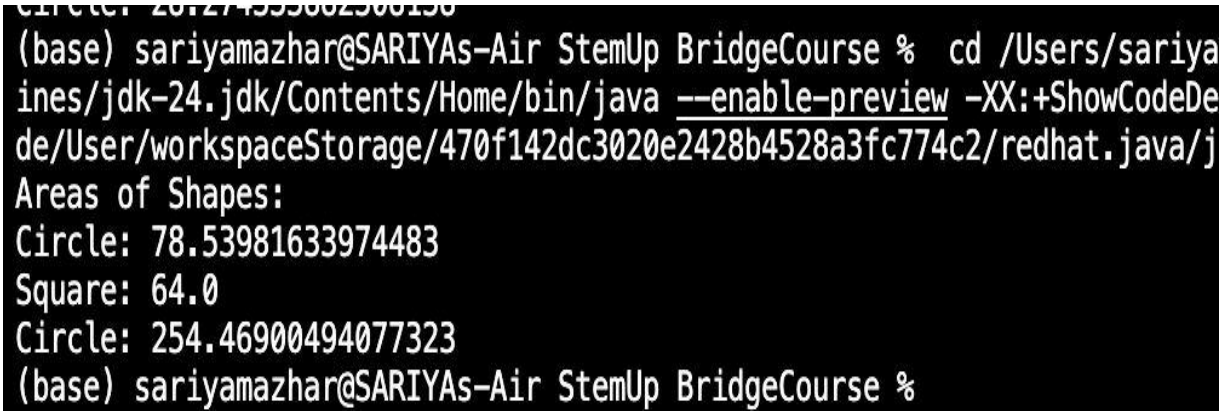
```
public class Shapes {
    public static void main(String[] args) {

        Shape[] shapes = {
            new Circle(5),
            new Square(8),
            new Circle(9)
        };



        System.out.println("Areas of Shapes:");
        for (Shape s : shapes) {
            System.out.println(s.getClass().getSimpleName() + ": " + s.getArea());
        }
    }}
```

5. Output



6. Observation

- This program demonstrates abstraction and polymorphism by using a common Shape reference to calculate areas of different shapes.It allows uniform handling of Circle and Square objects while providing shape-specific area computations.

Problem Solving Activity 10

1. **Program Statement:Tool Demo using polymorphism**.

- Base: Tool, method draw() Subclasses: PenTool, EraserTool, LineTool Demonstrate polymorphism using a collection

---

2. Algorithm

Step 1: Define a base class Tool with a method draw()

Step 2: Create subclasses PenTool, EraserTool, and LineTool, each overriding the draw() method

Step 3: Initialize a collection (e.g., ArrayList) of type Tool

Step 4: Add objects of PenTool, EraserTool, and LineTool to the collection

Step 5: Iterate over the collection and call draw() on each object

Step 6: Observe polymorphic behavior as each object executes its own version of draw()

---

3. **Pseudocode**

```
class Tool
    method draw()
        print "Using a generic tool
class PenTool inherits Tool
    override method draw()
        print "Drawing with Pen Tool"
class EraserTool inherits Tool
    override method draw()
        print "Erasing with Eraser Tool"
class LineTool inherits Tool
    override method draw()
        print "Drawing a Line with Line Tool"
create list tools
add PenTool to tools
add EraserTool to tools
add LineTool to tools
```

```
        for each tool in tools
            tool.draw()
```

---

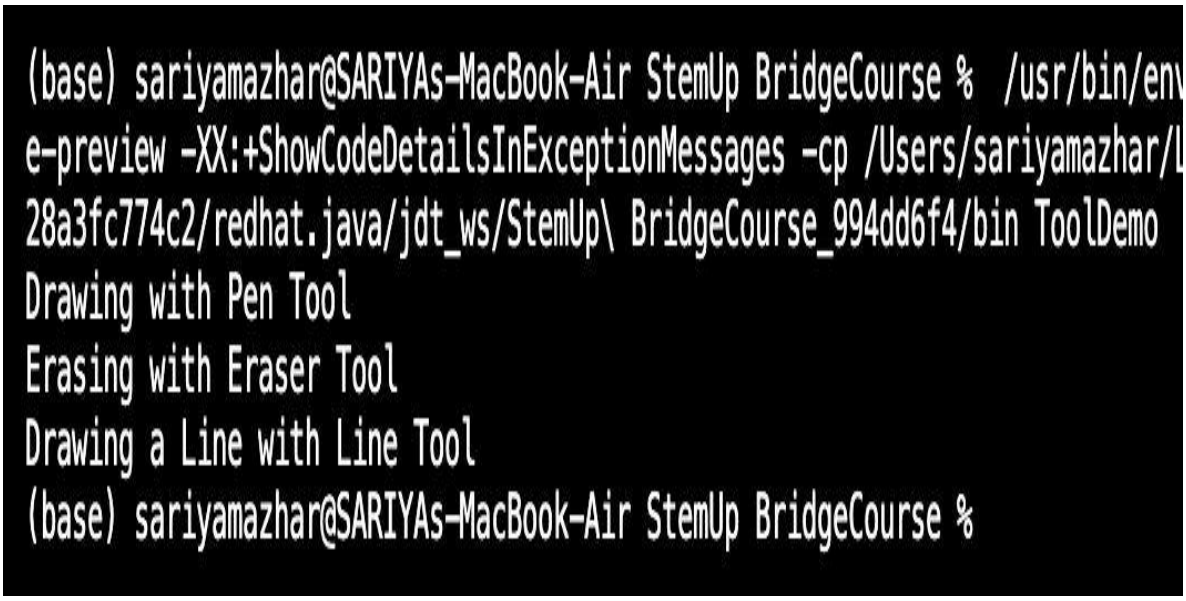
4. Code

```
import java.util.*;

// Base class
class Tool {
    void draw() {
        System.out.println("Using a generic tool");
    }
}
class PenTool extends Tool {
    void draw() {
        System.out.println("Drawing with Pen Tool");
    }
}
class EraserTool extends Tool {
    void draw() {
        System.out.println("Erasing with Eraser Tool");
    }
}
class LineTool extends Tool {
    void draw() {
        System.out.println("Drawing a Line with Line Tool");
    }
}
public class ToolDemo {
    public static void main(String[] args) {
        // Create a collection of Tool references
        ArrayList<Tool> tools = new ArrayList<>();
```

```
        tools.add(new PenTool());
        tools.add(new EraserTool());
        tools.add(new LineTool());
        for (Tool tool : tools) {
            tool.draw(); // Calls appropriate overridden method
        }}}
```

5. Output



6. Observation

- Polymorphism allows the draw() method to behave differently based on the actual object type. Using a collection of base class references simplifies managing multiple tool types.