UNIVERSITY OF MALAYA

WIX1002 : FUNDAMENTALS OF PROGRAMMING

GROUP MEMBERS :
JITESH A/L MOGANA RAJA (25006745)
TAN CHEE KEAT (25006123)
TEH XU ZHE (25006355)
LEE MING DAO (25006825)
LIM HONG ZHANG (25006100)

FACULTY: FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

TITLE: VIVA 3 REPORT

SEMESTER SESSION: SEM I 2025/2026

LECTURER NAME : DR. NURUL BINTI JAPAR

GROUP NAME : Ctrl+C Ctrl+V

Question 1:

1.1 Problem Description

The task is to define a Role class representing a character in a guild-based fantasy setting. Each Role has the following attributes:

- name (String): Name of the role.
- age (Integer): Age of the character (can be null).
- race (String): Race of the character (e.g., "Human", "Elf", "Orc").
- mana (Double): Magic energy of the character.

The requirements are:

1. Implement a no-argument constructor that sets default values.
2. Implement a parameterized constructor to initialize all attributes.
3. Provide getter and setter methods for all variables to ensure encapsulation.
4. Implement a method performAction() that prints a basic action message for the role.

The Role class itself is a blueprint and does not produce output when run on its own.

1.2 Solution Explanation

To solve this problem:

1. Encapsulation:
    - All instance variables are declared private to protect the internal state of the object.
    - Public getter and setter methods are provided to allow controlled access and modification.
2. Constructors:

- The no-argument constructor initializes the role with default values: "Unknown" for name and race, null for age, and 0.0 for mana.

- The parameterized constructor allows creation of Role objects with custom values for all attributes.

3. Behavior Method:

    - The performAction() method outputs a message indicating the role is performing a magical action.

    - This method uses the name of the role to personalize the output.

4. Usage:

    - While the Role class itself produces no output, it can be tested by creating objects in another class (e.g., RoleTest) and calling performAction().

1.3 Sample Input and Output

The class definition file in Question 1 contains the blueprint for the object but produces no direct output upon execution as it lacks a main method. The sample input and output demonstrating its functionality are instead provided via the RoleTest class in Question 2.

1.4 Source Code

```
package com.guild.roles.Role;

/**
 *
 * @author xuzhe
 */
public class Role {

    // Instance variables
    private String name;    // Name of the role
    private Integer age;    // Age (can be null)
    private String race;    // Race (e.g., "Human", "Elf", "Orc")
    private Double mana;    // Mana (magic energy)

    // No-argument constructor (default values)
    public Role() {
        this.name = "Unknown";
        this.age = null;
```

```
    this.race = "Unknown";
    this.mana = 0.0;
}

// Parameterized constructor
public Role(String name, Integer age, String race, Double mana) {
    this.name = name;
    this.age = age;
    this.race = race;
    this.mana = mana;
}

// Getter and Setter for name
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

// Getter and Setter for age
public Integer getAge() {
    return age;
}

public void setAge(Integer age) {
    this.age = age;
}

// Getter and Setter for race
public String getRace() {
    return race;
}

public void setRace(String race) {
    this.race = race;
}

// Getter and Setter for mana
public Double getMana() {
    return mana;
}
```

```
    public void setMana(Double mana) {
        this.mana = mana;
    }

    // Method to perform an action
    public void performAction() {
        System.out.println(name + " is performing a magical action.");
    }
}
```

Question 2:

2.1 Problem Description

The task is to create a test class RoleTest to demonstrate the functionality of the Role class defined in Question 1.

Requirements:

1. Create several Role objects using both the no-argument and parameterized constructors.
2. Modify some attributes of the roles using setter methods.
3. Call the performAction() method for each object to show their actions.
4. Print out all attributes (name, age, race, mana) of each role to verify that the data is correctly stored and updated.

This class is responsible for generating output, unlike the Role class which only defines the structure and behavior of the object.

2.2 Solution Explanation

To solve this problem:

1. Object Creation:
    - Create multiple Role objects using the no-argument constructor to test default values.
    - Create objects using the parameterized constructor to test custom initialization.
2. Modify Attributes:
    - Use setter methods (setName(), setAge(), setRace(), setMana()) to update attributes of the roles created with default constructors.
3. Perform Actions:

- Call the performAction() method for each role object to demonstrate the role performing a magical action.

4. Print Role Details:

   - Define a helper method printRole(Role player) to display all attributes of each role.

   - This method uses getter methods (getName(), getAge(), getRace(), getMana()) to access encapsulated data.

2.3 Sample Input and Output

```
run:
Ali is performing a magical action.
ABU is performing a magical action.
Pikachu is performing a magical action.
Jitesh is performing a magical action.
Name:Ali
Age :21
Race:Human
Mana:14.0
Name:ABU
Age :25
Race:Human
Mana:17.0
Name:Pikachu
Age :100
Race:Elf
Mana:10.0
Name:Jitesh
Age :20
Race:Human
Mana:8.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

2.4 Source Code

```
package com.guild.roles.Role;

/**
 *
 * @author xuzhe
 */
public class RoleTest {
    public static void printRole(Role player){
        System.out.println("Name:"+player.getName());
        System.out.println("Age :"+player.getAge());
        System.out.println("Race:"+player.getRace());
        System.out.println("Mana:"+player.getMana());
    }

    public static void main(String[] args) {
    //Create several Role objects using both constructors
    Role p1=new Role();
    Role p2=new Role();
    Role p3=new Role("Pikachu",100,"Elf",10.0);
    Role p4=new Role("Jitesh",20,"Human",8.0);

    //Use setter methods to modify some attributes
    p1.setName("Ali");
    p1.setAge(21);
    p1.setRace("Human");
    p1.setMana(14.0);

    p2.setName("ABU");
    p2.setAge(25);
    p2.setRace("Human");
    p2.setMana(17.0);

    //Call performAction() for each object
    p1.performAction();
    p2.performAction();
    p3.performAction();
    p4.performAction();

    printRole(p1);
```

```
  printRole(p2);
  printRole(p3);
  printRole(p4);
  }
}
```

QUESTION 3;

3.1 Problem Statement

Background

In the world of **MapleStory**, Evan is preparing for a decisive battle against the Black Mage. To survive the onslaught, he decides to craft a custom magical defense: the **MagicShield**. The effectiveness of this shield relies on its physical dimensions and its elemental alignment.

Your task is to implement a Java class representing this shield, adhering to strict encapsulation and logic requirements.

Package Definition

- **Package Name:** com.maplestory
- **Class Name:** MagicShield

Class Requirements:

1. **State (Private Fields):**
    - Double radius (meters), Double thickness (cm), String elementType.
    - **Static:** int shieldCount (tracks total objects), constants DEFAULT_RADIUS (1.0), DEFAULT_THICKNESS (5.0).
2. **Constructors:**
    - **No-Arg:** Sets fields to defaults and "Neutral".
    - **Parameterized:** Sets fields to inputs.
    - *Note:* Both must increment shieldCount.
3. **Encapsulation & Validation:**
    - Setters for radius/thickness must throw IllegalArgumentException if values are negative.
    - setElementType defaults null/empty inputs to "Neutral".
4. **Behavior (Formulas):**
    - **Defense Power:** (radius×thickness)×Coefficient

        - *Coefficients:* Fire (1.1), Ice (1.2), Light (1.3), Dark (1.4), Neutral (1.0).
    - **Mana Cost:** (radius×10)+(thickness×2).
5. **Static Utilities:**
    - isValidSize(Double size): Checks if size is non-null and positive.
    - Static versions of defense and mana calculation methods.

3.2 Solution Explanation

The solution utilizes **Wrapper Classes** (Double) to handle potential null states and strictly separates object state from calculation logic.

Key Implementation Details:

- **Validation:** Logic is centralized in *Setters* to ensure data integrity during both construction and modification.
- **Logic Reuse:** Mathematical formulas are implemented as static methods. Instance methods simply delegate to these static methods, preventing code duplication.
- **Polymorphism:** A switch statement handles the mapping of elemental strings to numerical coefficients.

3.3 Sample Input and Output

The class definition file in Question 3 contains the blueprint for the object but produces no direct output upon execution as it lacks a main method.
The sample input and output demonstrating its functionality are instead provided via the **MagicShieldTest class** in Question 4.

3.4 Source Code

```
package com.maplestory;

/**

 *

 * @author jitesh

 */

public class MagicShield {

    // --- 1. Static Variables (Shared/Constants) ---

    private static final double DEFAULT_RADIUS = 1.0;

    private static final double DEFAULT_THICKNESS = 5.0;

    private static int shieldCount = 0;



    // --- 2. Instance Variables (Encapsulated State) ---

    // Using Double wrapper class to allow 'null' values as per
requirements

    private Double radius;

    private Double thickness;

    private String elementType;



    // --- 3. Constructors ---

    // No-argument constructor

    public MagicShield() {

        this.radius = DEFAULT_RADIUS;
```

```
        this.thickness = DEFAULT_THICKNESS;

        this.elementType = "Neutral";

        shieldCount++; // Increment global counter

    }

    // Parameterized constructor

    public MagicShield(Double radius, Double thickness, String
elementType) {

        this.radius = radius;

        this.elementType = elementType;

        this.thickness = thickness;

        shieldCount++; // Increment global counter

    }



    // --- 4. Encapsulation Methods (Getters & Setters) ---

    public Double getRadius() {

        return radius;

    }

    public void setRadius(Double radius) {

        // Validation: must not be negative

        if (radius != null && radius < 0) {

            throw new IllegalArgumentException("Invalid radius");

        }

        this.radius = radius;

    }
```

```
    public Double getThickness() {
        return thickness;
    }
    public void setThickness(Double thickness) {
        // Validation: must not be negative
        if (thickness != null && thickness < 0) {
            throw new IllegalArgumentException("Invalid thickness");
        }
        this.thickness = thickness;
    }

    public String getElementType() {
        return elementType;
    }
    public void setElementType(String type) {
        // Validation: handle null or empty strings
        if (type == null || type.trim().isEmpty()) {
            this.elementType = "Neutral";
        } else {
            this.elementType = type;
        }
    }
    // --- 5. Class Methods (Static Logic) ---
```

```
    public static int getShieldCount() {
        return shieldCount;
    }
    public static boolean isValidSize(Double size) {
        return size != null && size >= 0;
    }
    // Static calculation to compute defense for ANY input
    public static double calculateDefensePower(double r, double t, String
type) {
        double coefficient;

        // Normalize string to avoid case sensitivity issues (optional but
recommended)
        String standardizedType = (type == null) ? "Neutral" : type;
        coefficient = switch (standardizedType) {
            case "Fire" -> 1.1;
            case "Ice" -> 1.2;
            case "Light" -> 1.3;
            case "Dark" -> 1.4;
            default -> 1.0;
        }; // Covers "Neutral" and unknown types

        return (r * t) * coefficient;
    }
```

```
    // Static calculation to compute mana for ANY input
    public static double calculateManaCost(double r, double t) {
        return (r * 10) + (t * 2);
    }
    // --- 6. Instance Methods (Object Logic) ---
    public double calculateDefensePower() {
        // Safety check: if fields are null, return 0 or handle gracefully
        if (this.radius == null || this.thickness == null) return 0.0;

        // Re-use the static logic to avoid code duplication
        return calculateDefensePower(this.radius, this.thickness,
this.elementType);
    }
    public double calculateManaCost() {
        if (this.radius == null || this.thickness == null) return 0.0;

        // Re-use the static logic
        return calculateManaCost(this.radius, this.thickness);
    }
    /**
     *
     * @return String format*/
    @Override
    public String toString(){
```

```
        return String.format(
                        "[MagicShield Info]%n" +
                        "Element Type: %s%n" +
                        "Radius: %.1f m%n" +
                        "Thickness: %.1f cm%n" +
                        "Defense Power: %.2f%n" +
                        "Mana Cost: %.1f%n",
                this.elementType, this.radius, this.thickness,
calculateDefensePower(),calculateManaCost()
        );
    }
}
```

QUESTION 4:

4.1 Problem Description

This problem focuses on designing and testing a Java class named MagicShield using object-oriented programming concepts.
The class represents a magical shield with properties such as radius, thickness, and element type.

The task requires:

- Creating objects using different constructors
- Applying encapsulation with getters and setters
- Handling invalid input values using exceptions
- Using instance methods to calculate defense power and mana cost
- Using static methods and variables to perform calculations without creating objects and to track the total number of shields created
- Testing edge cases such as null, zero, and negative values

A separate test class (MagicShieldTest) is used to verify that all functionalities work correctly and follow proper Java programming practices

4.2 Solution Explanation

To solve the problem, the MagicShield class is designed with private instance variables to ensure encapsulation.
Both a no-argument constructor and a parameterized constructor are provided to allow flexible object creation. Default values are used when no arguments are supplied.

Setter methods are implemented with validation logic. If invalid values such as negative radius or thickness are provided, the program throws an IllegalArgumentException. This ensures that the object always remains in a valid state.

Instance methods are used to calculate defense power and mana cost based on the current properties of each shield object.
Static methods are included to allow calculations using arbitrary values without creating an object, demonstrating proper use of class-level behavior.

A static counter tracks the total number of MagicShield objects created. This verifies the correct use of static variables shared across all instances.

The test class creates multiple objects, modifies values using setters, handles exceptions using try-catch blocks, and prints results using the overridden toString() method. Edge cases such as zero, negative, and null values are also tested to ensure robustness and reliability of the solution.

4.3 Sample Input & Output

```
=== MagicShield Test ===

[MagicShield Info]
Element Type: Neutral
Radius: 1.0 m
Thickness: 5.0 cm
Defense Power: 5.00
Mana Cost: 20.0

[MagicShield Info]
Element Type: Fire
Radius: 3.0 m
Thickness: 6.0 cm
Defense Power: 19.80
Mana Cost: 42.0

[MagicShield Info]
Element Type: Ice
Radius: 2.5 m
Thickness: 4.0 cm
Defense Power: 12.00
Mana Cost: 33.0

=== Setter Tests ===
fireShield radius updated: 5.0
Caught exception for invalid radius: Invalid radius.
iceShield thickness updated: 3.5
Caught exception for invalid thickness: Invalid thickness.
iceShield element type after null set: Neutral

=== Instance Method Calculations ===
fireShield Defense Power: 33.00
fireShield Mana Cost: 62.0
iceShield Defense Power: 8.75
iceShield Mana Cost: 32.0

=== Static Method Calculations ===
Arbitrary Shield -> Defense Power: 28.00, Mana Cost: 50.0

Total shields created: 3

=== Edge Case Tests ===
[MagicShield Info]
Element Type: Neutral
Radius: 0.0 m
Thickness: 0.0 cm
Defense Power: 0.00
Mana Cost: 0.0

Caught exception for negative shield creation: Invalid radius.
Static check for size 0: true
Static check for negative size -3: false
BUILD SUCCESSFUL (total time: 0 seconds)
```

4.4 Source Code

```
package com.maplestory;

/**
 *
 * @author xuzhe
 */
public class MagicShieldTest {

    public static void main(String[] args) {

        System.out.println("=== MagicShield Test ===\n");

        // Create multiple MagicShield objects using different constructors
        MagicShield defaultShield = new MagicShield(); // no-arg constructor
        MagicShield fireShield = new MagicShield(3.0, 6.0, "Fire"); // parametric constructor
        MagicShield iceShield = new MagicShield(2.5, 4.0, "Ice");

        // Print initial shields
        System.out.println(defaultShield);
        System.out.println(fireShield);
        System.out.println(iceShield);

        // Test setters with valid and invalid values
        System.out.println("=== Setter Tests ===");

        try {
            fireShield.setRadius(5.0); // valid
            System.out.println("fireShield radius updated: " + fireShield.getRadius());
            fireShield.setRadius(-2.0); // invalid
        } catch (IllegalArgumentException e) {
            System.out.println("Caught exception for invalid radius: " + e.getMessage());
        }

        try {
            iceShield.setThickness(3.5); // valid
            System.out.println("iceShield thickness updated: " + iceShield.getThickness());
            iceShield.setThickness(-1.0); // invalid
        } catch (IllegalArgumentException e) {
            System.out.println("Caught exception for invalid thickness: " + e.getMessage());
        }

        iceShield.setElementType("Light"); // valid
        iceShield.setElementType(null);    // should default to Neutral
```

```
        System.out.println("iceShield element type after null set: " + iceShield.getElementType());

        //  Calculate and print defense power and mana cost
        System.out.println("\n=== Instance Method Calculations ===");
        System.out.printf("fireShield Defense Power: %.2f%n",
fireShield.calculateDefensePower());
        System.out.printf("fireShield Mana Cost: %.1f%n", fireShield.calculateManaCost());

        System.out.printf("iceShield Defense Power: %.2f%n",
iceShield.calculateDefensePower());
        System.out.printf("iceShield Mana Cost: %.1f%n", iceShield.calculateManaCost());

        // Use static methods to calculate shield properties for arbitrary parameters
        System.out.println("\n=== Static Method Calculations ===");
        double dp = MagicShield.calculateDefensePower(4.0, 5.0, "Dark");
        double mana = MagicShield.calculateManaCost(4.0, 5.0);
        System.out.printf("Arbitrary Shield -> Defense Power: %.2f, Mana Cost: %.1f%n", dp,
mana);

        // Output total number of shields created
        System.out.println("\nTotal shields created: " + MagicShield.getShieldCount());

        // Test edge cases: 0, negative, null
        System.out.println("\n=== Edge Case Tests ===");
        MagicShield zeroShield = new MagicShield(0, 0, "Neutral");
        System.out.println(zeroShield);

        try {
            MagicShield invalidShield = new MagicShield(-1, -5, null);
            System.out.println(invalidShield);
        } catch (IllegalArgumentException e) {
            System.out.println("Caught exception for negative shield creation: " + e.getMessage());
        }

        // Test static utility with negative / zero
        System.out.println("Static check for size 0: " + MagicShield.isValidSize(0));
        System.out.println("Static check for negative size -3: " + MagicShield.isValidSize(-3));

    }
}
```

QUESTION 5:

5.1 Problem

Package: com.magical.people.Person

Variables:

Instance Variables (Encapsulated)

- private String name – Student name
- private int age – Student age (null indicates unknown age)

Static Variables

- private static final int DEFAULT_AGE = 18 – Default age, used for no-arg constructor
- private static int personCount = 0 – Tracks total number of students in the academy

Constructors:

No-argument constructor

- Initialize name = "Unknown Student"
- Initialize age = DEFAULT_AGE
- Increment personCount

Parameterized constructor

- Accept name and age parameters and assign them to fields
- Increment personCount

Encapsulation Methods

- getName() / setName(String name)
- getAge() / setAge(int age)
    - o Validate that age is not negative; if negative, throw IllegalArgumentException

Class Methods

- static int getPersonCount() – Returns the total number of students created
- public boolean compareTo(Person other)
    - o Returns true if both name and age are identical between the current student and other; otherwise, returns false

5.2 Solution

1. Create the Person class
2. Define the base class for students within the package com.magical.people.
3. Declare instance variables:
    a. private String name
    b. private int age
4. Declare static variables:
    a. private static final int DEFAULT_AGE = 18
    b. private static int personCount = 0 (Increments whenever a new student is created)
5. Create an empty (no-arg) constructor:
    a. Initializes name = "Unknown Student"
    b. Initializes age = DEFAULT_AGE
    c. Increments personCount
6. Create a parameterized constructor:
    a. Accepts name and age as arguments.
    b. Increments personCount
7. Create mutator (setters) and accessor (getters) methods:
    a. Includes validation in setAge: if the age is less than 0, throw an IllegalArgumentException.
8. Define the compareTo method:
    a. Receives another Person object as an argument.
    b. Returns true only if both the name and age are identical.

5.3 Sample Input and Output

The class definition file in Question 5 contains the blueprint for the object but produces no direct output upon execution as it lacks a main method.

The sample input and output demonstrating its functionality are instead provided via the **PersonTest class** in Question 6.

5.4 Source Code

```
package viva3.com.magical.people;

public class Person {
    private String name;
    private int age;

    private static final int DEFAULT_AGE = 18;
    private static int personCount = 0;

    public Person() {
        this.name = "Unknown Student";
        this.age = DEFAULT_AGE;
        personCount++;
    }

    public Person(String name, int age) {
        this.name = name;
        setAge(age);
        personCount++;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0)
            throw new IllegalArgumentException("Age cannot be negative");
        this.age = age;
    }

    public static int getPersonCount() {
        return personCount;
    }
```

```
    public boolean compareTo(Person other) {
        if (other == null)
            return false;
        return this.name.equals(other.name) && this.age == other.age;
    }
}
```

QUESTION 6:

6.1 Problem

Package: com.magical.people.PersonTest

Requirements :

Create multiple Person objects using both no-arg and parameterized constructors

Call setAge() with valid and invalid ages

→verify that exceptions are thrown for invalid input

Call compareTo() to compare different students

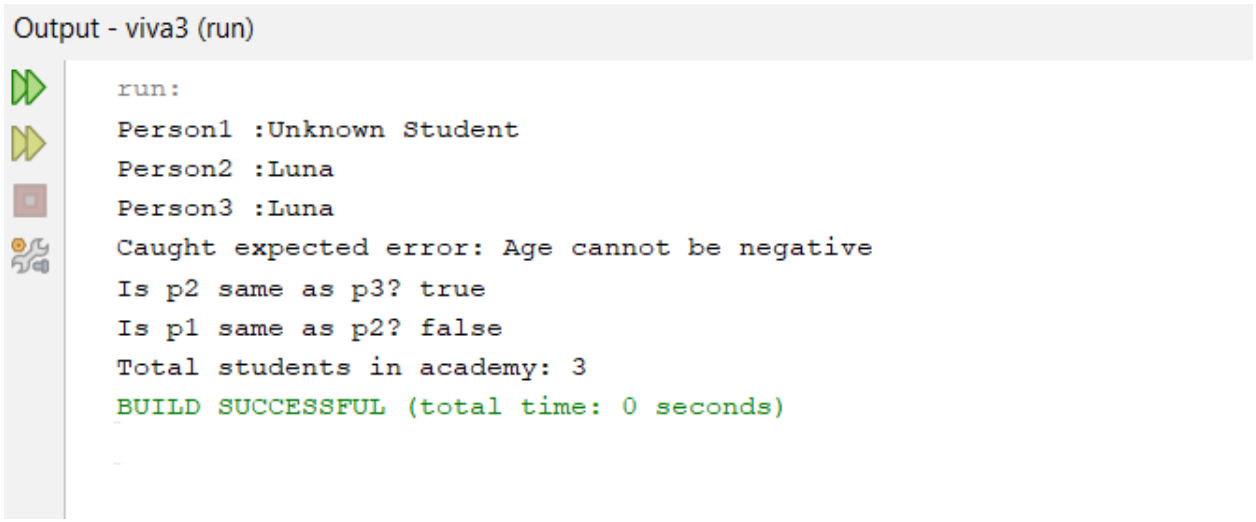
→output whether they are exactly the same

Call getPersonCount() to verify that the total student count in the academy is correct

6.2 Solution

1. Create the PersonTest class
2. Create multiple Person objects: Use both the empty constructor and the parameterized constructor.
3. Test validation logic: Use a try-catch block to attempt setting a negative age and verify the exception.
4. Compare students: Use the compareTo method to check if two student objects are identical.
5. Display totals: Print the static personCount to ensure the registry is tracking students accurately.

6.3 Sample Input and Output



6.4 Source Code

```
package viva3.com.magical.people;

public class PersonTest {
    public static void main(String[] args) {
        // Create objects
        Person p1 = new Person();
        Person p2 = new Person("Luna", 20);
        Person p3 = new Person("Luna", 20);

        String person1=p1.getName();
        String person2=p2.getName();
        String person3=p3.getName();

        System.out.println("Person1 :"+person1);
        System.out.println("Person2 :"+person2);
        System.out.println("Person3 :"+person3);

        // Test exceptions
        try {
            p1.setAge(-5);
```

```
        } catch (IllegalArgumentException e) {
            System.out.println("Caught expected error: " + e.getMessage());
        }

        // Test compareTo
        System.out.println("Is p2 same as p3? " + p2.compareTo(p3)); // Should be true
        System.out.println("Is p1 same as p2? " + p1.compareTo(p2)); // Should be false

        // Verify count
        System.out.println("Total students in academy: " + Person.getPersonCount());
    }
}
```