CST-239 Activity 3 Guide

Contents

Part 1: Person Interface

Overview

Goal and Directions:

In this activity, you will learn how implement **Interface** classes, as well as implement the **comparable** interface on the Person class implemented in Activity #2. Complete the following tasks for this activity:

Execution

Execute this assignment according to the following guidelines:

1. Create a Person Interface class:
   a. Create a new Java Project named *topic3-1*. Copy the *Person* and *Test* classes from Activity #2 (select files from *topic2-3* project, use Ctrl-C, then go to *topic3-1* project *src* folder, and use Ctrl-V, then refactor package name accordingly for *topic3-1*).
   b. Create a new Java Interface named *PersonInterface* in the *app* package. Interface classes are created by using New → Interface menu options.
   c. Create the following behavior methods:
      a. public void walk();
      b. public void run();
      c. public boolean isRunning();

```
public interface PersonInterface
{
    public void walk();
    public void run();
    public boolean isRunning();
}
```

   d. Update the *Person* class so it implements the *PersonInterface* interface class.

```
public class Person implements PersonInterface
```

e. Notice that Eclipse says you must add unimplemented methods to the *Person* class. If you automatically add the unimplemented methods, the new code should appear at the bottom of the file.

f. Implement each of the *PersonInterface* methods in the *Person* class, providing a simple console print statements for the walk and run methods to give some feedback and change the status of the walking and running properties.

```
public void walk()
{
    System.out.println("I am walking");
    running = false;
}
public void run()
{
    System.out.println("I am running");
    running = true;
}
public boolean isRunning()
{
    return running;
}
```

g. Update the *Test* class to test out the new *walk*, *run*, and *isRunning* methods.

```
// Make a Person walk and run
person1.walk();
person1.run();
System.out.println("Person 1 is running: " + person1.isRunning());
person1.walk();
System.out.println("Person 1 is running: " + person1.isRunning());
```

h. Run the *Test* class. Take a screenshot of the output.

2. Implement Comparable Interface:

a. Update the *Person* class so it implements the *compareTo<Person>()* interface class.

```
public class Person implements PersonInterface, Comparable<Person>
```

b. Notice that Eclipse says you must add unimplemented methods to the *Person* class. If you automatically add the unimplemented methods, the new code should appear at the bottom of the file.

c. Implement the *Comparable* Interface by implementing the *public int compareTo<Person>()* method in the *Person* class that will compare a person by their last name with logic checks for supporting if the last name is the same.

```
@Override
public int compareTo(Person p)
{
    int value = this.lastName.compareTo(p.lastName);
    if(value == 0)
    {
        return this.firstName.compareTo(p.firstName);
    }
    else
    {
        return value;
    }
}
```

d. Update the *Test* class that creates some people (of type *Person*) and sorts them by last name. Note, you can use the Arrays.sort() method from *java.util* package to implement your sort algorithm. This sort function will call your *compareTo()* method! Reorder your people array multiple times to prove that your *compareTo()* method works properly.

```
// Create a bunch of Persons and compare them so they are sorted on Last Name
Person[] persons = new Person[4];
persons[0] = new Person("Justine", "Reha");
persons[1] = new Person("Brianna", "Reha");
persons[2] = new Person("Mary", "Reha");
persons[3] = new Person("Mark", "Reha");
Arrays.sort(persons);
for(int x=0;x < 4;++x)
{
    System.out.println(persons[x]);
}
```

e. Run the *Test* class. Take a screenshot of the output.

f. Add an *age* property of type *int* to the *Person* class. Reimplement your *compareTo()* method to compare a person's age.

g. Run the *Test* class. Take a screenshot of the output.

h. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

i. Generate the JavaDoc for the *Person* class, *PersonInterface* class, and the *Test* class.

Deliverables:

The following need to be submitted as this part of the activity:

a. Theory of operation write-ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 2: Polymorphic Shapes

Overview

Goal and Directions:

In this activity, you will implement **Interface** classes and also **Polymorphic** classes for various shapes. In this assignment, organize your classes into **packages**. Packages are simply folders to help you organize the code. Complete the following tasks for this activity:

Execution

Execute this assignment according to the following guidelines:

1. Create a new project called *topic3-2*.
2. Create a Shape Interface class:
    a. Create an Interface class named *ShapeInterface* in a *base* package. Use the File → Interface menu options to create the Interface class.
    b. Define a single method *public int calculateArea()* in the Interface class.

```
public interface ShapeInterface
{
    int calculateArea();
}
```

3. Create a Super/Base Class:
    a. Create a new Java Class *ShapeBase* that implements the *ShapeInterface* in the *base* package.
    b. Implement the calculateArea() method that returns a -1.
    c. Implement three protected class member variables *width* and *height* of type int and a *name* of type String.
    d. Implement a non-default constructor that is passed a *name*, *width*, and *height* as arguments. Use the non-default constructor to initialize the protected class member variables.
    e. Implement a *getter* (i.e., getName()) method for the *name* class member variable.

```java
public class ShapeBase implements ShapeInterface
{
    protected String name;
    protected int width, height;

    public ShapeBase(String name, int width, int height)
    {
        this.name = name;
        this.width = width;
        this.height = height;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int calculateArea()
    {
        return  -1;
    }
}
```

4. Implement Triangle and Rectangle Shapes:
    a. Create a new class named *Triangle* that inherits from the *ShapeBase* class in the *shape* package. Implement a non-default constructor that is passed in a *width* and *height* as argument that also calls the super/base class constructor. Implement the *calculateArea()* that calculates and returns the area for a triangle.

```java
public class Triangle extends ShapeBase
{
    public Triangle(String name, int width, int height)
    {
        super(name, width, height);
    }

    @Override
    public int calculateArea()
    {
        return width * height/2;
    }
}
```

    b. Create a new class named *Rectangle* that inherits from the *ShapeBase* class in the *shape* package. Implement a non-default constructor that is passed in a *width* and *height* as argument that also calls the super/base class constructor. Implement the *calculateArea()* that calculates and returns the area for a rectangle.

```java
public class Rectangle extends ShapeBase
{
    public Rectangle(String name, int width, int height)
    {
        super(name, width, height);
    }

    @Override
    public int calculateArea()
    {
        return width * height;
    }
}
```

5. Implement a *Test* class in the *test* package with a *main()* method:
    a. Create a private helper method named *displayArea()* that takes a *ShapeBase* as an argument and prints the shape's name and area.

```
private static void displayArea(ShapeBase shape)
{
    System.out.println("This is a shape named " + shape.getName() + " with an area of " + shape.calculateArea());
}
```

    b. In the *main()* method:
        i. Create method scoped variable that is an array of type *ShapeBase* named *shapes* that can hold 2 shapes.
        ii. Initialize the first array element with an instance of a *Rectangle* class.
        iii. Initialize the second array element with an instance of a *Triangle* class.
        iv. Loop over the *shapes* array and call the private *displayArea()* helper method for each shape.

```
public static void main(String[] args)
{
    // Create an array of Base Shapes and initialize to specific Shapes
    ShapeBase[] shapes = new ShapeBase[2];
    shapes[0] = new Rectangle("Rectangle", 10, 200);
    shapes[1] = new Triangle("Triangle", 10, 50);

    // For all Shapes display its area
    for(int x=0;x < shapes.length;++x)
    {
        displayArea(shapes[x]);
    }
}
```

        v. Take a screenshot of the final output.
6. Extension:
    a. Create two new classes, each representing one of these shapes: Circle, Oval, Regular Hexagon, or Trapezoid.
    b. Implement the *calculateArea()* method for both of the new classes.
    c. Create instances of the new classes and add them to array used in *Test* class.
    d. Run the *Test* class and take a screenshot of the final output.
7. Write up:
    a. Draw a UML Class Diagram of your solution.
    b. In 3–4 sentences, describe where and how polymorphism was demonstrated in your code.
    c. Generate the JavaDoc for all classes.

Deliverables:

The following need to be submitted as this part of the activity:

a. UML diagram of solution.
b. Theory of operation write-ups.
c. All screenshots of application in operation.

d. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 3: Polymorphic Weapons

Overview

Goal and Directions:

In this activity, you will learn refactor the code from Activity #2 to use **Interface** classes and demonstrate **Polymorphism** for various Weapons used in a game. Complete the following tasks for this activity:

Execution

Execute this assignment according to the following guidelines:

1. Create a new project called *topic3-3*. Copy all the code from Activity #2 (select files from *topic2-2* project, use Ctrl-C, then go to *topic3-3* project *src* folder and use Ctrl-V, then refactor package name accordingly for *topic3-3*).
2. Refactor the base *Weapon* class:
   a. Remove the *Weapon* abstract class and create a new Interface class named *WeaponInterface*.
   b. Add a *public void fireWeapon()* method that returns void and takes no arguments.
   c. Add a *public void fireWeapon()* method that returns void and takes a power argument as an integer type.
   d. Add a *public void activate()* method that returns void and takes an argument as a boolean type.

```
public interface WeaponInterface
{
    public void fireWeapon();
    public void fireWeapon(int power);
    public void activate(boolean enable);
}
```

3. Refactor the specialization Weapon Classes:
   a. Refactor the *Bomb* class so that it implements the *WeaponInterface* class. Implement the methods from the *WeaponInterface*. The implementation can simply print to the console the class name, method name, and any method arguments.

```
public class Bomb implements WeaponInterface
{
    @Override
    public void fireWeapon(int power)
    {
        System.out.println("In Bomb.fireWeapon() with a power of " + power);
    }

    @Override
    public void fireWeapon()
    {
        System.out.println("In Bomb.fireWeapon()");
    }

    @Override
    public void activate(boolean enable)
    {
        System.out.println("In the Bomb.activate() with an enable of " + enable);
    }
}
```

b. Refactor the *Gun* class so that that it implements the *WeaponInterface* class. The implementation can simply print to the console the class name, method name, and any method arguments.

```
public class Gun implements WeaponInterface
{
    @Override
    public void fireWeapon(int power)
    {
        System.out.println("In Gun.fireWeapon() with a power of " + power);
    }

    @Override
    public void fireWeapon()
    {
        System.out.println("In Gun.fireWeapon()");
    }

    @Override
    public void activate(boolean enable)
    {
        System.out.println("In the Gun.activate() with an enable of " + enable);
    }
}
```

4. Refactor the Game Class:
   a. Remove all existing game logic in the *main()* method.
   b. Create a private helper method *fireWeapon*() that takes a *WeaponInterface* as an argument as a weapon. For the passed in weapon, activate the weapon then fire the weapon.

```
private static void fireWeapon(WeaponInterface weapon)
{
    if (weapon instanceof Bomb)
        System.out.println("----------> I am a Bomb");

    weapon.activate(true);
    weapon.fireWeapon(5);
}
```

   c. Create a method scoped variable that is an array of type *WeaponInterface* named *weapons* that can hold 2 weapons.
   d. Initialize the first array element with an instance of a *Bomb* class.
   e. Initialize the second array element with an instance of a *Gun* class.

f. Loop over the *weapons* array and call the private *fireWeapon()* helper method for each weapon.

```
public static void main(String[] args)
{
    // Create an array of WeaponInterface and initialize to specific Weapon of Bomb and Gun
    WeaponInterface[] weapons = new WeaponInterface[2];
    weapons[0] = new Bomb();
    weapons[1] = new Gun();

    // For all Weapons fire them
    for(int x=0;x < weapons.length;++x)
    {
        fireWeapon(weapons[x]);
    }
}
```

g. Take a screen shot of the final output.

5. Write up:
   a. Draw a UML Class Diagram of your solution.
   b. In 3–4 sentences, describe where and how polymorphism was demonstrated in your code.
   c. Generate the JavaDoc for all classes.

Deliverables:

The following need to be submitted as this part of the activity:

a. UML diagram of solution.
b. Theory of operation write-ups.
c. All screenshots of application in operation.
d. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 4: Practice Using the Debugger

Overview

Goal and Directions:

In this activity, you will continue to practice using the debugger. Complete the following tasks for this activity:

Execution

Pick a class out that was coded in this activity and, following steps outlined in Part 4 of Activity 1, demonstrate with the code in this activity the abilities to set a breakpoint, inspect variables, step into code, and inspect the call stack.

Deliverables:
The following need to be submitted as this part of the activity:

a. Screenshot from the Setting Breakpoints task.
b. Screenshots from the Inspecting Variables task.
c. Screenshots from the Stepping task.
d. Screenshot from the Inspecting Call Stack task.

Research Questions

1. Research Questions: Online students will address these in the Discussion Forum and traditional on ground students will address them in this assignment.
    a. What are some reasons a programmer might decide to declare a class as abstract? What are some reasons a programmer might decide to declare a class as an interface? Summarize your answers and rationale in 300 words.
    b. What does the keyword final do on a class member variable? What does the keyword final do on a class method? What does the keyword final do on a class? Why would you want to mark a class or method as static? Summarize your answers and rationale in 400 words.

Final Activity Submission

1. In a Microsoft Word document, complete the following for the Activity Report:
    a. Cover sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the diagrams, screenshots, and theory of operation write-ups.
    c. Zip file with all code and generated JavaDoc documentation files.
    d. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report and zip file of the code and documentation to the Learning Management System (LMS).