Step 9 – Interface, polymorphism, casting

Last week we learned about abstraction. I mentioned that we can perform abstraction in two ways in Java: Abstract classes and interface. So this time it's the interface's turn. But when we work with abstractions, it becomes completely natural to learn about polymorphism and casting as well!

Objectives for this step:

- I can use the interface, and I understand what it means.
- I understand what polymorphism is.
- I know how I can use casting.

Relevant chapters in the book:

- Chapter 11: Using interfaces
- Chapter 15: Polymorphism explained
- Chapter 16: Polymorphism in action

Here is the questionnaire for step 9 where you can tell how it went.

We will continue with the example that deals with geometric shapes. I have made a starting point you can use (see github repo), but you can also choose to take your starting point in your own code, from the previous exercise. It must be possible to move the figures. This means that they must also have information about where they are. For a circle, we only need information about one point (center of the figure). For rectangles and squares, we need to know the location at two points (for example, top-left corner and bottom-right corner). Points have a location in a coordinate system with values for x and y. Figures must be able to be moved up, down, right and left.

Task 1

Add the interface *Movable*. This interface will have the following methods:

- *moveUp(double distance)*
- *moveDown(double distance)*
- *moveRight(double distance)*
- *moveLeft(double distance)*

Task 2

Add a class *MovablePoint* containing two fields: *x* and *y* (both of type double).

The class must implement Movable. moveUp should increase y by the specified value, moveDown should decrease y by the specified value, and correspondingly for movements in the x-direction (moveRight / Left). If you need some help, here is a hint:

When a class needs to implement an interface, we write <ClassName> implements <InterfaceName>.

Do you need an example of one of the methods? Then you get this:

```
public void moveUp(double distance) {

    y = y + distance; //or y += distance;

}
```

Remember to override the toString method as well.

Task 3

We want all our figures to have a position in a coordinate system. We therefore want to know where on the x-axis and y-axis the figures are located. One challenge is that the characters are different. For circles, we only need to know one point (the center of the circle). For rectangles and squares, we need to know the location at two points (for example, top-left corner and bottom-right corner).

Let us try to use the class MovablePoint from problem 2.

Use MovablePoint when making sure a circle has a position. If you're thinking "Hey, how the hell can I know how to do that?!?" Then you can get some white tips on a white background below.

Task 4

Make sure that the Rectangle and Square shapes also have a location. Remember that both must have two points: one for the top-left corner of the figure, and one for the bottom-right.

Task 5

We want all **geometric shapes to be Movable**. Do you have any idea how we can do that? Chew on it a bit….

Have you finished thinking? Absolutely sure?

I think it might be wise to let Shape implement Movable. All geometric shapes are a Shape. Then it must be better to let Shape implement Movable than to let all the subclasses of Shape do it?

Modify Shape to implement the *Movable* interface.

What happened?

You will see that we do not get any compilation errors in the Shape class. It is abstract. But the subclasses are having problems. We are now saying that all Shapes must implement the methods from the Movable interface. But that is not the case (yet).

So how do we fix it? What do you think?

It would have been great to be able to implement the methods in Shape. Because then it could apply to all subclasses. But we have a problem with some figures having one point for position, and some having two. Let us therefore try to implement the methods in Circle first.

Change Circle so that the methods from the Movable interface are implemented. Difficult? Then you get help from white text on a white background below. But try to figure it out on your own first😊

Task 6

Make sure both Rectangle and Square implement the *Movable* interface.

Task 7

Whoops, there's nothing more to do now, is there? Yes, we have to test if this works. And then we can, at the same time, look at some uses of polymorphism! Use the Test class to create methods that test the code.

Create one object of each type (Circle, Rectangle and Square). Put these in an ArrayList.

Hmm, the objects are of different types. What type should we use when creating the ArrayList? Think about it, and get white-text help below if necessary.

Task 8

Create a unique method for each type (Circle, Rectangle and Square) – i.e. a method that only exists in the individual classes. Example for Circle: public void uniqueCircleMethod () {<SOUT something>}.

Create one object of each type, but this time you will upcast the object references when the objects are created. For example:

Shape s1 = new Circle(1, Color.PINK, true, new MovablePoint(0.0, 0.0));

Check which methods you have access to via the object references. Do you have access to the unique methods?

Put the objects in an ArrayList. Go through all the objects in the list. For each object, print (SOUT) area and perimeter information. Then call on the unique method of the object (hint: instanceof).

Extra Task 1

Since we have defined two corners for rectangles and squares here (topLeft and bottomRight), it might be appropriate to validate that the objects created by these types comply with certain rules for placement. For example, it does not make sense for a topLeft corner to be located to the right of bottomRight. Therefore, make sure that objects are created only if the location of the corners makes sense.

Extra Task 2 (Extra tricky)

Fair warning.

Same idea as the previous extra assignment: validate that the objects are consistent. In addition to validating the location, you must validate that the lengths of the sides match the location.

And if you're done with everything, a reminder for more exercises:

code wars

Kattis