Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19

Part I
Programming in Java

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica I

Polimorfism in Java: inheritance and overloading

Contents

1 Introduction

In this practice you will define classes by reusing other classes under the view of software reusability. Several implementation options are proposed leading to more stable solutions when facing possible changes of the problem.

1.1 The problem

We are asked to design a class to store information from two types of geometric figures: circles and triangles. So we need first to define one class for each type of figure.

The `Circle` class is defined with two attributes `x` and `y` in order to specify where this shape is located in a two-dimensional space. An additional attribute `radius` stores the radius of the circle. The `Triangle` class is defined with two coordinates, as with the circle, together with the attributes `base` and `height` which are related to its size. These definitions are shown in Figure 1 together with a constructor and `equals(Object)` and `toString()` methods.

```
public class Circle {
 private double x, y;
 private double radius;

 public Circle(double a,
        double b, double c){
   x = a; y = b; radius = c;
 }

 public boolean equals(Object o){
  if (!(o instanceof Circle)) {
     return false; }
  Circle c = (Circle) o;
  return x == c.x && y == c.y &&
  radius == c.radius;
 }

 public String toString(){
  return ``Circle:\n\t'' +
    ``Position:  ('' +x+ ``,'' +
    y+``)\n\tRadius:  ''  +radius;
 }
}
```

```
public class Triangle {
  private double x, y;
  private double base, height;

  public Triangle(double a,
     double b, double c, double d){
   x = a; y = b;
   base = c; height = d;
  }
  public boolean equals(Object o){
    if (!(o instanceof Triangle)){
       return false; }
    Triangle t = (Triangle) o;
    return x == t.x && y == t.y &&
           base == t.base &&
           height == t.height;
  }
  public String toString(){
    return ``Triangle:\n\t'' +
      ``Position:  ('' +x+ ``,'' +
      y+``)\n\tBase:  ''  +base+
      ``\n\tHeight:  ''  +height;
  }
}
```

Figure 1: Definition of the `Circle` and `Triangle` classes, without inheritance

NOTE: Method overloading. Remark that the implementation of the `equals(Object)` and `toString()` methods constitute an overloading of the same methods, inherited from the `Object` class.[1]

Since the header of the `equals` method does not restrict the input parameter, it is recommended the use of `instanceof` in order to deal with the possibility of receiving an instance of a different class. Let us remember that we can ask in Java if an instance of a class is from a particular type by using `instanceof` with the following syntax:

```
variableReferenceToObject instanceof NameOfTheClass
```

In this way, the first instruction of the implementations of the `equals` method of Figure 1 allows us to return the value `false` when the received object is not a circle or a triangle, respectively. The second instruction (**explicit coercion** or casting to `Circle` and to `Triangle`, respectively) is required in order to have a reference to those classes and, in this way, to be able to access their attributes (which would not be possible with a reference of type `Object`). Remark that this casting operation is safe because it is only performed after the checking made by means of `instanceof`.

Importance of casting.
Reflect on the potential problems posed by the following implementation of the method for the `Circle` class:

```
public boolean equals(Object o) {
  return x == ((Circle)o).x && y == ((Circle)o).y &&
         radius == ((Circle)o).radius;
}
```

Now that we have the definition of the two classes representing the two types of figures, we should think about how to store a group of figures that can be triangles or circles.

We could pose a solution by using an array of elements of type `Object` and by making use of type coercion and checking in order to ensure a correct operation. However, this solution is not recommended from the point of view of software engineering (maintainability, extensibility, etc.). More details in the third year course.

Let us now see two solutions which make use of some language features (namely, inheritance and polymorphic variables).

[1] There is a hierarchy of already defined classes in Java. The root of this hierarchy is the `Object` class, and this is the reason why any Java class inherits from `Object`.

2 Solution 1: Making use of inheritance and overloading

Inheritance allows us to define a class hierarchy by grouping common features and behavior in a common *parent* class. In our problem, we can observe that both circles and triangles are placed in a position in the plane. Thus, it makes sense to create a new class `Figure` to characterize the position and to inherit from it the two types of figure. The definition of this class is shown in Figure 2. It contains two attributes `x` and `y` of type `double`, a constructor method as well as `equals(Object)` and `toString()` methods.

```
public class Figure {
  private double x, y;
  public Figure(double x, double y) {
    this.x = x; this.y = y;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Figure)) { return false; }
    Figure f = (Figure) o;
    return x == f.x && y == f.y;
  }
  public String toString() {
    return ``Position: ('' + x + ``, '' + y + ``)'';
  }
}
```

Figure 2: `Figure` class

Now, we can extend the `Figure` class with the `Circle` class. Put in other words, `Circle` will inherit from `Figure`:

```
public class Circle extends Figure {
  private double radius;
  ...
}
```

Since the attributes of the `Figure` are private, they are not visible from other classes, including the derived classes. A possible way to define the constructor of `Circle` (since we do not have access to their attributes `x` and `y`), is to use the constructor of the base class (the `Figure` class) by means of the reserved word `super`. The constructor would be as follows:[2]

```
public Circle(double x, double y, double r) {
  super(x, y);
  radius = r;
}
```

[2]The complete code is shown in Figure 3.

When the constructor of the base class is invoked, this call must be the first instruction of the body of the subclass constructor method.

> **NOTE:** The reserved word `super` can also be used to make references to methods of the base class that have been overwritten in the subclass, for instance: `super.toString();`

Figure 3 illustrates the complete code of the `Circle` and `Triangle` classes defined as subclasses of `Figure`.

```
public class Circle
   extends Figure {
 private double radius;

 public Circle(double x,
  double y, double r) {
    super(x, y);
    radius = r;
 }

 public String toString() {
  return ''Circle:\n\t'' +
         super.toString() +
         ''\n\tRadius: '' +
         radius;
 }
}
```

```
public class Triangle
   extends Figure {
 private double base, height;

 public Triangle(double x,
  double y, double b, double h) {
    super(x, y);
    base = b;
    height = h;
 }
 public String toString() {
  return ''Triangle:\n\t'' +
         super.toString() +
         ''\n\tBase: '' + base +
         ''\n\tHeight: '' + height;
 }
}
```

Figure 3: New definition of the `Circle` and `Triangle` classes

Visibility of attributes and reuse

The visibility of attributes and methods of a base class can be relaxed in order to make them visible for every derived class (even when they belong to a different package) by using the `protected` modifier instead of the `private` modifier.

An appropriate use of inheritance is to reuse as much as possible everything declared in the superclass. That's why the proposed solution invokes the constructor of the `Figure` class in the constructors of the subclasses (note that this is compatible with the declaration of attributes either as `protected` or as `private`) instead of assigning values to the inherited attributes (this last solution would only be possible if attributes were declared as `protected`).

The same principle can be applied for method overwriting: if what is implemented in the superclass can be applied in the subclass, it is preferable to invoke the superclass method. An example is the `toString()` method of the `Circle` and `Triangle` classes (see Figure 3).

Exercise 1 *The* `equals` *method defined in the* `Figure` *class establish that two figures are equal when they have the same position. We have to refine this method for the* `Circle` *and* `Triangle` *subclasses:*

Overwrite the `equals(Object)` *method in the* `Circle` *and* `Triangle` *subclasses by reusing what is already implemented in the* `Figure` *class.*

Defining the group of figures

Once the class hierarchy for the figures has been defined, we have to address the question of defining a group of figures able to store both triangles and circles. The `FiguresGroup` class (see Figure 4) makes use of an array of type `Figure`. This means that its components can only contain objects of this type and the corresponding derived types: `Circle` and `Triangle`.

```
public class FiguresGroup {
 private static final int NUM_FIGURES = 10;  // constante
 private Figure[] figuresList = new Figure[NUM_FIGURES];
 private int numF = 0;
 public void add(Figure f) { figuresList[numF++] = f; }
 public String toString() {
  String s = ‘‘’’;
  for(int i = 0; i < numF; i++) s += ‘‘\n’’ + figuresList[i];
  return s;
 }
}
```

Figure 4: The `FiguresGroup` class making use of type `Figure`

If we would need to distinguish the particular type of an object stored in the array, we would have to use the `instanceof` expression, but we do not need to care about the possibility of objects in the array not belonging to the `Figure` class (or any of its descendants). Even so, it might be possible to reference to objects of type `Figure` (as will be seen in the second solution, this can be avoided by making `Figure` an abstract class).

The following `FiguresGroupUse` is presented as an example of use of the `FiguresGroup` class:

```
public class FiguresGroupUse {
 public static void main(String[] args) {
  FiguresGroup g = new FiguresGroup();
  g.add(new Circle(10, 5, 3.5));
  g.add(new Triangle(10, 5, 6.5, 32));
  System.out.println(g);
 }
}
```

Figure 5: Definition of the `FiguresGroupUse` class

The result of the execution of the `FiguresGroupUse` class is shown below:

```
Salida Estándar
Circle:
    Position: (10.0, 5.0)
    Radius: 3.5
Triangle:
    Position: (10.0, 5.0)
    Base: 6.5
    Height: 32.0
```

Equality of groups of figures

Two groups of figures are considered to be equal if they contain the same figures no matter the order nor the number of times they appear in the group. That is, it suffices to check if every figure contained in one group is also contained in the other and vice versa.

In the previous implementation of the `FiguresGroup` class the method `equals(Object)` was not defined. In the following exercise you should implement this method for the new class (overloading). In order to simplify this task, the implementation of two auxiliary (private) methods is provided:

```java
public class FiguresGroup {
 ...
 private boolean found(Figure f) {
  for(int i = 0; i < numF; i++) {
     if (figuresList[i].equals(f)) return true;
  }
  return false;
 }
 private boolean included(FiguresGroup g) {
  for(int i = 0; i < g.numF; i++) {
     if (!found(g.figuresList[i])) return false;
  }
  return true;
 }
}
```

The `found(Figure)` method verifies if the figure received as argument can be found, or not, in the group of figures where this method is invoked. Remark that the use of `equals` in the expression `figuresList[i].equals(f)` can be interpreted as follows: if there is a reference to a `Circle` object in the position `i` of the array, then the `equals` method of the `Circle` class is invoked; however, if there is an object of type `Triangle`, the code of the `equals` method of the `Triangle` class would be executed instead. This decision will be taken at execution time: it is an example of polymorphism with dynamic binding.

The method `included(FiguresGroup)` checks if the group `g` received as argument is included or not in the group of figures that is inkoved in the method call (i.e. `this`). To this end, the method checks if each of the figures contained in `g` can be found in `this`.

Exercise 2 *Overwrite the* `equals(Object)` *method for the* `FiguresGroup` *class taking into account that you have the methods* `included(FiguresGroup)` *and* `found(Figure)` *at your disposal in the same class.*

Try the implemented method by invoking it in `FiguresGroupUse` *and by comparing different objects.*

Exercise 3 *Consider the changes that would be required in the methods* `add` *and* `equals` *of the* `FiguresGroup` *class if the groups were considered as sets (otherwise stated, if you can assume that there cannot be repeated elements). It is not necessary to make the actual implementation.*

Figure 6 illustrates the classes and their relationships. Classes are rep-

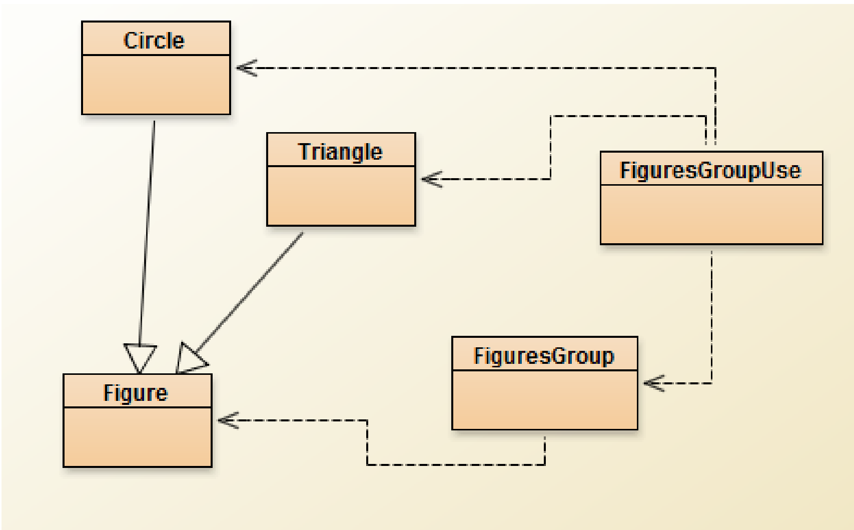

Figure 6: *IS-A* and *USES-A* relationships.

resented by means of boxes and their relationships by using different types of lines and arrows. There are two kinds of relations between classes: On the one hand, the inheritance relation *IS-A* is represented by a full line with a solid arrow from the derived class to the parent class. This relation establishes a hierarchy of classes where the parent class is more general that the derived one. On the other hand, the relation *USES-A* is represented by a dotted line. You can observe in Figure 6 a diagram where the class `FiguresGroup` makes use of `Figure`, `Circle` and `Triangle`, but

`FiguresGroup` is not derived from other class. The classes `Circle` and `Triangle` do not use any data type, just inherit from the type `Figure`. Finally, the class `FiguresGroupUse` uses the classes `Circle`, `Triangle` and `FiguresGroup` but it does not use the class `Figure`.

Exercise 4 *In order to represent the geometric figure rectangle, define a class* `Rectangle` *that inherits from* `Figure`*. A rectangle is a shape with two attributes of type* `double` *(*`base` *and* `height`*) and with the same methods as* `Circle` *and* `Triangle`*. Is there any change in* `FiguresGroup`*? Add a rectangle in the group of figures defined by the class* `FiguresGroupUse` *to check it works.*

Exercise 5 *In order to represent the geometric figure square, define the* `Square` *class, without attributes, derived from* `Rectangle` *and with the same functionality as its superclass. Did you need to overload any method of* `Rectangle`*? Why?*

3 Solution 2: Using abstract classes

We have seen in previous section how to restrict the types of objects that can be included in the `figuresList` array. Only objects of `Figure` class and its derived subclasses are allowed, but this also implies that an instance of `Figure` can be created and stored in the array as in the following example:

```
figuresList[pos] = new Figure(10, 5);
```

However, our initial purpose was to represent a group of geometric figures (circles, triangles, rectangles, squares and whatever geometric figure that will be created) but not objects of type `Figure`. A way to avoid this consists in making `Figure` an abstract class. Abstract classes are usually used to develop a class hierarchy with some common behavior.

```
public abstract class Figura {
 protected double x, y;
 ...
}
```

Observe how the modifier of attributes `x` and `y` have changed. This change allows these attributes to be visible from the derived classes. Remember that this is not always necessary: it will depend on how the classes are defined.

Exercise 6 *Observe again the* `FiguresGroup` *class in Figure 4. The inclusion of objets from the* `Figure` *class in the array of figures could be avoided by checking the type of object received by the* `add(Figure)` *method by means* *of* `instanceof` *in order to allow only the insertion of objects descending from* `Figure`*.*

This solution would be clearly worse and you do not have to implement it. ¿How this method will be affected when defining new types of figure (derived from `Figure`*)? Which advantage offer inheritance regarding the maintenance of the application in this case?*

Exercise 7 *Let us suppose we want a new method* `area()`*, available for any type of figure. Define an abstract method* `area()` *for the* `Figure` *class that returns a value of type* `double`*.*

After this modification of the `Figure` class, the derived classes will no longer compile and the following error would be observed: *"... is not abstract and does not override abstract method area() in Figure"*.

Exercise 8 *Solve this problem by implementing the* `area()` *method in each of the subclasses of* `Figure`*.*

Remember that hte area of a cirble can be computed by using the expression `Math.PI * Math.pow(radius, 2)`*, whereas the are of the triangle can be computed as* `base * height / 2`*, and the area of the rectangle by means of* `base * height`*.*

Modify the code of `FiguresGroupUse` *in order to compute the are of the created figures.*

Figure 7 illustrates the class hierarchy for the geometric figures defined so far. The `FiguresGroup` class uses the type `Figure` without needing to change its code and the test class `FiguresGroupUse` also remains unchanged using `FiguresGroup` and the classes derived from `Figure` to create objects of such classes.

Exercise 9 *Define a method called* `area()` *in the* `FiguresGroup` *class that computes and returns the sum of all the areas of shapes contained in the group of shapes. To this end, traverse all shapes references in the* `figuresList` *attribute from index* `0` *to* `numF-1` *applying the method* `area()` *to each one. As you can observe, inheritance provides method polymorphism since each instance makes use of the proper type of* `area()` *method given by its class.*

Exercise 10 *Define a method called* `greatestFigure()` *in the* `FiguresGroup` *class that returns the reference of the figure from the group whose area is the biggest one. Again, you will need to traverse each of the figures from the* `figuresList` *array and to apply the* `area()` *method to each figure.*

Additional exercises

The following exercises are proposed as an extension of this practice. They are considered an extension since they could extend the time provided in the laboratory session.

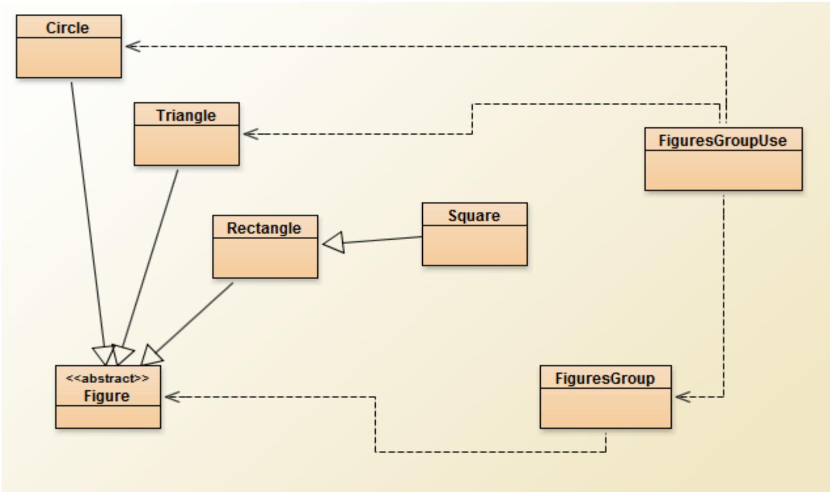

Figure 7: *IS-A* and *USES-A* relationships.

Exercise 11 *Now, it is required for every figure to have a method to compute its perimeter. Define an abstract method called* `perimeter()`*, returning a value of type* `double`*, in the* `Figure` *class.*

Exercise 12 *The method* `perimeter()` *must be implemented in every subclass of* `Figure`*. Remember that the perimeter of a circle can be computed by means of* `2 * Math.PI * radius`*, whereas the perimeter of a triangle is the sum of the lengths of its three sides, and the perimeter of a rectangle by means of* `2 * (base + height)`*. You can observe that we do not have enough information to compute the perimeter of a triangle. Which solution could you give to this problem? To declare* `Triangle` *as an abstract class? Not to compute the actual perimeter but to return an special value such as -1? To modify the declaration of the attributes of the class? Justify your choice.*