Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19

Part I
Programming in Java

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice II

Polimorfism and interfaces in Java

Contents

1 Objectives of this practical session

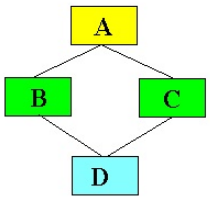
The **objective** of this second practice is to extend or enlarge the solution of the previous practice in order to make use of *interfaces*.

Before starting the practice exercises, the **concept of interface** is introduced, in Section 2, along with an explanation of its utility, its differences with abstract classes and their syntax. Section 3 proposes **exercises** to practice some of the issues addressed in the previous section. Your practice teacher can expand, if required, the exercises to be solved in the practice sessions.

2 Introduction

2.1 What is an interface

The concept of inheritance has been explained, in theory sessions, as a type of *Universal Inclusion Polimorfism*. You have also studied Abstract Classes and their rationale and purpose in *Java*. The exercises of the first practice made use of inheritance and abstract classes. An abstract class is a class from which no object can be directly instantiated and that may contain abstract methods. This kind of methods can be seen as a series of requirements for their derived classes. The inheritance mechanism used by these classes is simple: each class only inherits from another one, but one class may have many derived classes. Interfaces add some degree of flexibility to inheritance in *Java* because any class (even the interfaces themselves) can inherit from several interfaces. This feature increases the capability of poliformism of the language.



Interfaces introduce, to some extent, something similar to *multiple inheritance*. The general idea of this type of inheritance consist in allowing a single class to inherit from several ones. This might produce some problems. An example is what is known as the *diamond problem*, which appears when two classes `B` and `C` inherit a method `m` from a class `A` and adapt the inherited method by means of overwriting. If another class `D` inherits the two versions of the method `m` from the classes `B` and `C`, which version should use an instance of `D`? There are many programming languages with multiple inheritance (such as *C++*, *Eiffel*, *Python*, *Ruby* ...), and each one implements its own policy to solve this problem.

Interfaces are the most similar thing to multiple inheritance available in *Java*, although the problems inherent to this type of inheritance are not solved. These classes are similar to a pure abstract class where all methods are abstract and public (with no need to declare them explicitly as such), and

their attributes are static and final (constants). This implies that interfaces do not have constructors and we cannot instantiate objects from an interface. When a class `C` inherits from an interface `I`, we say that `C` ***implements*** `I`.

2.2 What can Java interfaces do for us?

An *Abstract Data Type* (ADT) is a set of values and operations that complies with the principles of abstraction and information hidding and that can be managed without knowing its internal representation: they are independent of the implementation. In other words, an ADT allows us to separate the specification of a class (what it does) from its implementation (how it does it). The use of ADT's results in more robust and less error prone programs. Interfaces are classes that can be used to specify an ADT. The classes implementing an interface, as well as their derived classes, extend their functionality but are forced to implement the set of abstract methods inherited from the interface. These implementations have to use the data structure of the class. We know that the methods of the interface can be used with the objects of the classes implementing this interface.

When defining interfaces we also allow the existence of *polymorphic variables* defined with the type of an interface. This allows the polymorphic invocation of methods on these variables. This restricts the use of objects in a class that implements an interface to the functionality specified in the interface.

Another feature of the interfaces is that they allow us to declare *constants* that will be available for all the classes implementing them. This saves code by avoiding the need to write the same constant declarations in different classes. At the same time, they are concentrated in a single place, which supposedly improves the maintenance of the code.

An additional feature is the *documentation* included in the interface. The syntax itself defines which methods are to be included in a class to meet the interface. But additional documentation is required related to the common features that the methods of the classes implementing it should actually do. In fact, when an interface is implemented, we have to conform to a *standard*. For example, if we want the objects of a class to be compared to put them in an order, the `Comparable` interface defined in the Java API is implemented. This requires the implementation of the `compareTo()` method, where the method profile sets the rule that the method must return a value of type `int`, whereas the interface documentation sets how it should work: return 0 if they are equal and a negative or positive integer depending on which object of the two compared is greater. Many predefined classes implement this interface.

There are some advisable rules in the use of the interfaces, among them we highlight two:

- Respect the *principle of segregation*: Classes derived from a class that implements an interface should not depend on interfaces that it does not use.
- Avoid the *interface pollution*. This happens when you add a method to a base class simply because some of the derived classes use it.

2.3 Differences between an interface and an abstract class

Syntactically, an interface is a completely abstract class, that is, it is simply a list of methods not implemented that can include the declaration of constants.[1] An abstract class can include implemented, abstract, constant, and variable methods.

An abstract class is used when we want to define an abstraction that encompasses objects of the different classes that inherit from it. In this way, it is possible to use polymorphism in the same "vertical" hierarchy of classes. An interface allows to choose which classes within one or different class hierarchies incorporate a certain extra functionality. Thus, interfaces do not force a hierarchical relationship, they simply make it possible, for classes not necessarily related, to have similar features, characteristics or behavior.

2.4 Syntax

Definition of an interface. An interface can extend from several interfaces but not from any class. Although the methods specified by an interface are abstract, the modifier `abstract` is **not** specified because all methods are abstract by default. They can not be private nor `protected` either.[2]

```
[visibilityModifier] interface interfaceName [extends interfaceList]
   {
    [CONSTANTS]
    [ [modifier] returnedType methodName1([parameters]);
                    ...
      [modifier] returnedType methodNameN([parameters]); ]
   }
```

Exercise 1 *According to the previous definition, is it possible to define empty interfaces? Do you know any of them?*

[1] Java 8 already allows you to include default methods and static methods in the interfaces, but in this practice we assume the use of an earlier version of Java.

[2] The brackets indicate optionality, and the `interfaceList` is a list of comma-separated names of interfaces.

Implementation of an interface. A class can only derive from a base class (`extends`), but it can implement multiple interfaces by typing their names separated by a comma after the reserved word `implements`.

```
[modifiersList] class ClassName [extends ClassName ]
                        [implements interfaceList]
   {
        ...
   }
```

3 Carrying out the practice

The exercises performed in this practice are the continuation of the first practice. Now, you will have to include additional functionalities to the classes in three steps: using a predefined interface, extending an interface and designing an interface.

The concept of genericity has been studied in the theory sessions and, specifically, its use in generic classes of *Java*. We will deepen into this concept in the next practice. In this current practice, generic classes are used without indicating their type arguments. When used in this way, they are known as *"raw" classes*, and the compiler will display a warning that a potentially dangerous use of the classes is being made since it will not be able to validate the types[3]. This feature is maintained in the latest versions of the language only to maintain compatibility with versions prior to *Java 5*.

3.1 Using a predefined interface

As discussed in the introduction of this practice, there is a commonly used interface for comparing with each other objects of a class: `Comparable`. This interface specifies the `int compareTo(Object)` method (the parameter is of type `Object` in its "raw" version). The rule states that the result of applying this method to an object `o1` receiving as parameter another object `o2` (that is, `o1.compareTo(o2)`) should be:

- if `o1` = `o2` the result of the comparison is the value zero.
- if `o1` < `o2` the result of the comparison is a negative integer value.
- if `o1` > `o2` the result of the comparison is a positive integer value.

This norm can be specified for each class that implements it. The comparison between figures does not have a natural order but you can define

[3]This warning can be removed by adding the `SuppressWarnings( "unchecked" )` expression in front of the method that uses generic classes in a "raw" form

an order among figures using their area (size). Thus, the norm for the comparison is adapted for the figures in the following way: given two figures `f1` and `f2` of a type derived from `Figure` as follows:

```
f1.compareTo(f2)
```

returns the following values:

- if `f1.area()` = `f2.area()` the result is the value zero.
- if `f1.area()` < `f2.area()` the result is a negative integer value.
- if `f1.area()` > `f2.area()` the result is a positive integer value.

Exercise 2 *Make the necessary changes to the* `Figure` *class so that it can be determined when one figure is larger than another. This class must implement the* `Comparable` *interface so that all the objects of its derived classes are comparable to each other.*

An alternative to the proposal of the previous exercise would be to implement the interface only in concrete classes. That is, each figure will implement its own `compareTo` method. If it is decided that the parameter of the method can only be of the type of the class where it is implemented, and we use `instanceof` to verify it, then only objects of the same class can be compared. This would prevent us from comparing two figures of any kind. To solve this problem we could check that the object received by this method is of the same type as the class where it is implemented, that is, to check that it is of type `Figure`. But this would lead to methods identical to the one you have implemented in the previous exercise spread across all concrete classes. The solution to the exercise you have performed is desirable for a good design (more sustainable from the point of view of maintenance of the application).

It was also commented, in section 2.2, that one of the uses of the interfaces consists in the specification of ADT's. The `List` interface is defined in the *java.util Java* package. This interface specifies the operations that a class must implement in order to see its elements as a list. These lists have a variable size and their elements occupy positions numbered consecutively with integers starting from 0. Some of the operations of this interface are:

- `void add(int index, Object element)` which inserts the element of the second parameter into the position indicated by the first parameter. `IndexOutOfBoundsException` is one of the exceptions that this method can throw. It is thrown for the case where the position does not exist.

- `void add(Object element)` works like the previous one but the element is added to the end of the list.
- `Object get(int index)` applied to a list returns the element occupying the position indicated by its parameter. If such a position does not exist it throws the exception `IndexOutOfBoundsException`.
- `int size()` returns the number of items in the list.

The classes `LinkedList` and `ArrayList` defined in the `java.util` package implement the `List` interface. Each one implements the previous operations taking into account its own data structures:

- `ArrayList`: Its implementation is based on a resizable array that doubles its size whenever more space is required.
- `LinkedList`: This implementation is based on a doubly linked list where each node has a reference to the previous and the next node.

In order to handle the elements of the `FiguresGroup` class as a list, this class should implement the `List` interface. This would involve writing the code of all methods as the `ArrayList` and `LinkedList` classes do. Another less costly alternative is to define a public method that provides an already implemented list. Next we will focus on the second option to practice with variables whose type is that of an interface. In addition, the implementations of `Comparable` that you have made in the previous exercise will also be used.

Exercise 3 *You have to implement a method in the* `FiguresGroup` *class that, when applied to a group of figures, returns an object that can be seen as a list of figures ordered by their area and in increasing order. The* insertion sort *algorithm must be used, and the profile of the method should be:* `public List orderedList()`. *The following steps can help you to solve the exercise using the operations of the* `List` *interface specified above and the comparator of the class* `Comparable`*:*

- *Add the required code to import the* `List` *interface and the* `LinkedList` *and* `ArrayList` *classes to the file where the* `FiguresGroup` *class is defined (that is, you have to import the* `java.util` *package).*
- *Write the profile of the* `orderedList` *method. In the body of this method you must create a list of figures where the figures that are present in the* `figuresList` *attribute will also be stored, but now in a sorted way. Remember that the* `numF` *attribute indicates the number of figures in the group and that they are stored in the* `figuresList` *array (in the low positions of the array up to the* `numF-1` *position). The body of the method you can implement it in the following way:*

- *Create an instance of a list using one of the two classes (namely,* `LinkedList` *or* `ArrayList`*. Choose only one of them). Now assign the reference of the created object to a variable of type* `List`*.* Observe that, from this variable, the compiler only knows that the methods defined in the `List` interface can be applied*. This variable will contain a reference to the object returned by this method. Add, if it exists, the first element of the* `figuresList` *array to the list of figures. To this end, you can use the* `add(Object)` *method from the* `List` *interface.*
- *in order to implement the direct insertion algorithm, make a loop performing an ascending traversal of the* `figuresList` *array starting from position* 1 *(because the element at position zero, if it exists, has already been inserted to the list). Remember that* `numF` *indicates the number of figures stored in the array.*
  Insert the rest of the figures from the array into the sorted list by implementing a ***search in a descending way*** *so that the list continues to be sorted. To insert each picture:*
  - *Define a new variable of type* `int` *to indicate the position of the element being processed in the list. Initialize it to the size of the list minus* 1*. Use the* `size()` *method to get the size of the list.*
  - *Write a downward loop to search in the list the position in which the figure is to be inserted. Note that at most, you have to reach the zero position (inclusive), and you have to ask if the figure being inserted, taken from the* `figuresList` *array, is smaller than the figure you are visiting on the list. Here you must use the* `get` *method of the* `List` *interface to get the figure of the list, and the* `compareTo` *method to compare it with the figure of* `figuresList` *array being inserted on the list.*
  - *The previous loop stops at a position prior to the insertion point, so you have to insert the figure from* `figuresList` *a position after the stop. You must use the* `add(int, Object)` *method of the* `List` *interface to perform the insertion in that position.*

Note*: No exceptions should be treated. We will allow them to propagate in case they occur.*

To check the proper operation of the `orderedList` method you can define a class `FiguresGroupUse` where you can create a group of figures with several figures. You can also use the `println()` method to display by standard

output the figures ordered by their areas. The list is converted into string by the `toString` methods implemented by the `ArrayList` and `LinkedList` classes which, in turn, make use of the `toString` methods implemented by each figure type. In the following `main` method, first the figures are inserted in the group in the order of calls to the `add` method, and then the figures are sorted in the group by using the `orderedList` method.

```
public static void main(String[] a) {
    FiguresGroup g = new FiguresGroup();
    g.add(new Circle(1.0, 6.0, 6.0));
    g.add(new Rectangle(2.0, 5.0, 10.0, 12.0));
    g.add(new Triangle(3.0, 4.0, 10.0, 2.0));
    g.add(new Circle(4.0, 3.0, 1.0));
    g.add(new Triangle(5.0, 1.0, 1.0, 2.0));
    g.add(new Square(6.0, 7.0, 15));
    g.add(new Rectangle(7.0, 2.0, 1.0, 3.0));
    System.out.println(g.orderedList());
}
```

3.2 Extending an interface

As discussed in Section 2, interfaces can only inherit from other interfaces. Derived interfaces add functionality to the specifications they inherit. The implementation of derived interfaces requires the implementation of the methods that are inherited. This is very useful when new methods are going to use methods that are inherited from the base class as in the next exercise.

Exercise 4 *Write a new interface named* `ComparableRange` *extending the* `Comparable` *interface by adding a method* `int compareToRange(Object o)`.

Exercise 5 *Make the class* `Rectangle` *implement the* `ComparableRange` *interface taking into account that only this comparison is used to compare pairs of rectangles and/or squares.*
The norm for this class is that it behaves similarly to its parent class except that it considers two figures equal if the difference of its areas (in absolute value) is less than or equal to 10% of the sum of its areas. If they are different under this criterion, they are compared the same as the `compareTo` *inherited method.*

To verify that the solution given to the above problem is correct, you can expand the `FiguresGroupUse` class by adding several rectangles and squares or random size to a list, and by looking for pairs of figures that are equal under the new equality criterion. For each pair that is equal you can show its position in the list, the name of their classes and their area in order to

verify that they are the same (under this criterion). It is an opportunity to choose an object of the type `ArrayList` or `LinkedList`: choose the one you have not used in exercise 3. The code shown in Figure 1 would be a possible implementation for obtaining random numbers by using `Random` class. Note that the expression `f.getClass().getName ()` is used to get the name of the class of the figure `f`.

```
List l = new LinkedList(); // O new ArrayList();
Random r = new Random();
for (int i = 0; i < 100; i++) {
    if (r.nextInt(2) == 0) {
        double b = r.nextDouble() * 10;
        double h = r.nextDouble() * 10;
        l.add(new Rectangle(1, 1, b, h));
    }
    else {
        double b = r.nextDouble() * 10;
        l.add(new Square(1, 1, b));
    }
}
for (int i = 0; i < 100; i++) {
    Rectangle fi = (Rectangle) l.get(i);
    for (int j = i + 1; j < 100; j++) {
        Rectangle fj = (Rectangle) l.get(j);
        if (fi.compareToRange(fj) == 0) {
            System.out.print("Figuras iguales: "
                + fi.getClass().getName() + " " + i
                + " y " + fj.getClass().getName() + " " + j
                + "\n Areas: " + fi.area()
                + ", " + fj.area() + "\n");
        }
    }
}
```

Figure 1: Code for checking the `compareToRange` method

3.3 Designing an interface

The use of the interfaces allows us to extend the functionality of classes that are not necessarily in the same class hierarchy. Indeed, they do not even need to be in the same package. In the following exercises you should create an interface to be implemented only by some of the classes in the hierarchy defined by the class `Figure` and its descendants.

```
int n = (int)radius;
for (int j = 0; j <= n * 2; j++) {
    for (int i = 0; i <= n * 2; i++) {
        if (Math.pow(i - n, 2.0) + Math.pow(j - n, 2.0)
            <= (int)Math.pow(n, 2)) {
            System.out.print(c);
        }
        else {
            System.out.print(" ");
        }
    }
    System.out.println();
}
```

Figure 2: Code to draw circles

```
int b = (int) base;
int h = (int) height;
for (int i = 0; i < h; i++) {
    for (int j = 0; j < b; j++) {
        System.out.print(c);
    }
    System.out.println();
}
```

Figure 3: Code to draw rectangles

Exercise 6 *Add the drawing functionality to some figures for which there is an algorithm to draw them in the terminal using a character. To this end, define a* `Printable` *interface that specifies the* `void print(char c)` *method. The class that implements this method must use the character received as parameter to draw its objects. Figures 2 and 3 illustrate the algorithms to draw rectangles and circles using their attributes:* `base` *and* `height` *for rectangles and* `radius` *for circles. The positions of these figures are not taken into account for drawing in the terminal.*

The classes that must implement the interface are those that descend from `Figure` *except* `Triangle` *and* `Square`*. The first one is excluded because there is no algorithm to draw it, whereas the second one is excluded because the* `print` *method inherited from* `Rectangle` *can also be used to draw squares.*

Exercise 7 *We want to draw all the "drawable" figures contained in a group of figures. Otherwise stated, the* `FiguresGroup` *class must implement the*

`Printable` *interface. If you try to use the following algorithm to draw all the figures in a group:*

```
public void print() {
    for (int i = 0; i < numF; i++) {
        figuresList[i].print();
    }
}
```

a compilation error occurs. You have to correct the code so that it compiles and works correctly. Note that the elements of `figuresList` *are of type* `Figure` *and that not all of them can be drawn. So, in addition to make sure that the class of a given figure implements* `Printable`*, you must allow the access to the* `print` *method which is not implemented in the* `Figure` *class (which is the type of the elements stored in the* `figuresList`*).*
In order to verify that it works correctly you can apply the `print` *method to the group defined in* `FiguresGroupUse`*.*