Programming challenges in Java

4. Classes and interfaces

Nacho Iborra

IES San Vicente

Table of Contents

Programming challenges in Java

1. Classes and inheritance

Java classes have more or less the same structure than in languages like C#. We use the class element to include all the elements of that class: attributes, constructors and methods. For instance, we can define a class called `Person` with the following code:

```
class Person
{
    String name;
    int age;

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

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }
}
```

Then, we can instantiate objects of this class, either separately or in an array or collection:

```
Person p = new Person("Juan", 20);

Person[] people = new Person[10];
people[0] = new Person("Ana", 33);
people[1] = new Person("Elena", 21);
...
```

1.1. Visibility

There are four visibility levels in the elements (attributes, constructors and methods) of a class:

- public: the element can be accessed from any other class
- protected: the element can be accessed from any subclass, or any class of the same package
- private: the element can only be accessed from current class
- package: this fourth visibility level is the default level when we don't specify any other one. It means that the element can be accessed from any other class of the same package.

Here you can see all these levels summarized:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

1.2. Inheritance

If we want to create a subclass from a given class, we use the extends modifier to refer to the parent class. For instance, we can create a class called `Child` that extends previous class `Person`. We add a new attribute with the school name, and its corresponding constructor and getter. See how to refer to the parent class (using super) from the constructor (or any other method):

```
class Child extends Person
{
    String school;

    public Child(String name, int age, String school)
    {
        super(name, age);
        this.school = school;
    }

    public String getSchool()
    {
        return school;
    }
}
```

The benefits and consequences of inheritance are the same than in other languages: subclass inherits everything from its parent class, and it can also add its own content.

2. Abstract classes and interfaces

2.1. Defining abstract classes

An abstract class is defined with the `abstract` modifier before the class name. It can also have abstract methods (although it is not compulsory), and it can't be instantiated directly. An abstract class is intended to define an incomplete code, that must be completed by the subclasses. Therefore, we should not instantiate a class that is not complete.

For instance, we can define an abstract class called Animal, to represent the information about any animal, such as its name. We can also define a method called "talk", but this method can't be implemented until we know the concrete type of animal that we are dealing with. So this class must be abstract, and "talk" method must be declared abstract as well.

```
public abstract class Animal
{
    String name;

    public Animal(String name)
    {
        this.name = name;
    }

    public abstract void talk();
}
```

Then, we can define a subclass, called Dog, that inherits from previous class. Then, we can override the abstract method and define its code:

```
public class Dog extends Animal
{
    public Dog(String name)
    {
        super(name);
    }

    @Override
    public void talk()
    {
        System.out.println("Wof wof");
    }
}
```

In the same way, we could define some other subclasses, such as Cat, or Cow. We can even define an abstract subclass, (Bird, for instance) to represent a whole branch of animals. This class should still be abstract, and it could also define its own abstract methods, such as "fly".

```
public abstract class Bird extends Animal
{
    public Bird(String name)
    {
        super(name);
```

```
    }

    public abstract void fly();
}
```

We can finally create some subclasses of this abstract class, such as Eagle. In this case, it must implement every abstract method that is not yet implemented from its superclasses.

```
public class Eagle extends Bird
{
    public Eagle(String name)
    {
        super(name);
    }

    @Override
    public void talk()
    {
        System.out.println("aaaaaaah");
    }

    @Override
    public void fly()
    {
        System.out.println("Eagle flying");
    }
}
```

2.2. Defining interfaces

An interface is just a set of methods that must be implemented by any class that wants to use this interfaces. This way, we grant that any class implementing a given interfaces will have any method of that interface. For instance, we can define an interface called Figure with a set of methods about that figure, such as calculating its area and its perimeter:

```
public interface Figure
{
    public double area();
    public double perimeter();
}
```

Then, we can define a class called Square that implements this interface, and then we must override these two methods inside this class:

```
public class Square implements Figure
{
    double side;

    public Square(double side)
    {
        this.side = side;
    }

    @Override
    public double area()
    {
        return side * side;
```

```
    }

    @Override
    public double perimeter()
    {
        return 4 * side;
    }
}
```

2.3. A practical example: sorting arrays or collections

In order to see a practical example of interface usage, let's face again a well known challenge: this one . If you don't remember it, you must sort a list of children according to two parameters:

- The *goodness level* (better children go first)
- The *weight* of their gifts (if two children have the same goodness level, then the one whose gifts are lighter goes first).

We have learnt how to solve this problem using a *quicksort* algorithm, but there is still a better way of solving this problem, that lets us forget how to implement the *quicksort.* We can implement an interface an call a simple Java method to sort an array automatically by any criteria.

We are going to create a class (called *Child*, for instance), with two attributes (goodness level and weight of the gifts), and a constructor and getters.

```
class Child
{
    int goodnessLevel;
    int weight;

    public Child(int goodnessLevel, int weight)
    {
        this.goodnessLevel = goodnessLevel;
        this.weight = weight;
    }

    public int getGoodnessLevel()
    {
        return goodnessLevel;
    }

    public int getWeight()
    {
        return weight;
    }
}
```

We need that this class can be compared, so that two different *Child* objects can be sorted. To do this, we can make this class implement *Comparable* interface, and then override the *compareTo* method. Inside this method, we will return a negative, zero or positive integer depending on whether this object is lower, equal or greater than the *Child* object received as a parameter.

```
class Child implements Comparable
{
    int goodnessLevel;
    int weight;
```

```
    public Child(int goodnessLevel, int weight)
    {
        this.goodnessLevel = goodnessLevel;
        this.weight = weight;
    }

    public int getGoodnessLevel()
    {
        return goodnessLevel;
    }

    public int getWeight()
    {
        return weight;
    }

    @Override
    public int compareTo(Object o)
    {
        Child otherChild = (Child)o;
        if (this.goodnessLevel > otherChild.goodnessLevel)
            return -1;
        else if (this.goodnessLevel < otherChild.goodnessLevel)
            return 1;
        else return this.weight - otherChild.weight;
    }
}
```

Our main class will create an array of *Child* objects with their corresponding goodness level and weight. Then, it will sort it by calling *Arrays.sort* method and show the data.

```
public class Challenge366
{
    public static void main(String[] args)
    {
        int numberOfChildren, goodnessLevel, weight;
        Child[] array;
        Scanner sc = new Scanner(System.in);
        do
        {
            numberOfChildren = sc.nextInt();
            sc.nextLine();

            if (numberOfChildren > 0)
            {
                array = new Child[numberOfChildren];

                for (int i = 0; i < numberOfChildren; i++)
                {
                    goodnessLevel = sc.nextInt();
                    weight = sc.nextInt();
                    sc.nextLine();
                    array[i] = new Child(goodnessLevel, weight);
                }

                Arrays.sort(array);

                for (int i = 0; i < numberOfChildren; i++)
                {
                    System.out.println("" + array[i].getGoodnessLevel() + " " +
                        array[i].getWeight());
```

```
                }
                System.out.println();
            }
        }
    while (numberOfChildren != 0);
    }
}
```

In this case, *Arrays.sort* method knows how to sort the array because we have told him how to do so by implementing the *Comparable* interface in the *Child* class.

2.4. Try yourself: Desarrollos en las bicicletas

Let's face this challenge. You should already know it, but this time you MUST solve it using classes and interfaces, as we have done with previous example.

2.5. Try yourself: Genética Jedi

This other challenge must be solved by using classes and interfaces as well.