Object-Orientation & Inheritance

Table of Contents

1. Object Orientation & Inheritance (II)

TOTAL: 90 min

1.1. Abstract classes

15 min 15 Send

Objective

Review Java syntax for abstract methods and abstract classes. Show some examples to know how and when to use dynamic binding with abstract methods.

Exercise

Until now, to calculate the album's price, the rate type (`NORMAL`, `REDUCED` or `INCREASED`) is checked inside the body of the `getPrice` method in the `Album` class. Depending on the rate type, we call the `getPrice` method of the related class: `NormalRate`, `ReducedRate` or `IncreasedRate`.

To simplify the program code, this check can be avoided by using dynamic binding. In order to do that, you must create an abstract `getPrice` method in `Rate` class and modify the current `getPrice` method in `Album` class. This method directly returns the price in `Rate` class multiplied by the number of elements in the `Album` class, without checking the rate type. As we are invoking an abstract method in Rate class, the runtime environment will know at runtime which the object type to which we refer is.

1.2. Exercise to practice with Figures

45 min 45 Send

In this exercise we are going to review the main concepts of Object Oriented Programming (OOP) in Java with an application to create a class hierarchy to define geometric figures.

IMPORTANT NOTE: Remember that it is essential that you test your code whenever you implement any method, even though you are not asked to do so explicitly.

The `Figure` interface.

In Java, an interface represents a kind of specification for all classes that implement it. Is is normally used to define a set of methods that we want to be provided by those classes.

For example, the Comparable interface represents objects that can be compared among them. It defines the `compareTo()` method that allows to compare two objects. Every class whose objects want to be compared, as stated before, must implement this method. With an interface, further design and reuse of code is easily made since all objects share the same comparison method and the result of it.

Before going on to the next sections, try to answer the following questions:

- What is an interface?
- How is it implemented in Java?
- What does it mean when a class *implements* an interface. How do you write this in Java?

All classes that are defined in this exercise must provide a basic set of methods, such as calculating the area of a geometric figure. In Java, this is achieved by defining the interface that declares all methods that must be implemented by the classes that implement it. For this exercise, every class that is created must implement the `Figure` interface.

The geometric figure.

An abstract class is a class that can not be instantiated. This is shown by the use of the `abstract` keyword in its declaration. An abstract class has "at least" one abstract method. That is, a method which is only declared but not implemented. This abstract method should be implemented by any subclass of the class. There is no point in instantiating objects of an abstract class and only subclasses of it can instantiate objects.

Remember that an abstract class can have constructors, though objects from it can not be instantiated.

1. Write the code of the `GeometricFigure` abstract class, which must store the common information for all geometric figures (for example, a descriptive label) and has to provide all methods that can be shared among figures and are independent from the concrete shape of them. This class must implement the `Figure` interface. Every class that can be represented by a geometric figure will inherit from `GeometricFigure` class.

   Every figure will have a descriptive label, so you should define an attribute of type `String` to hold the text of the label.

2. Write a constructor that receives as input parameter the descriptive label of the figure.

3. Write the `get` and `set` methods that allow to modify the text label's attribute. As these methods should not be modified by anybody, declare them as `final` to avoid any subclass to change the code of them (method overriding).

4. Implement the `printDescription()` method. Notice that is not an abstract method although it invokes other abstract methods. Also, to avoid subclasses to change the code of it (method overriding), declare it as `final`.

   The method must print a text description of the figure on the console, including the label's text, the type of figure and its area, with the following format:

   ```
   Tag: C-5
   Figure Type: Square
   Area: 25
   ```

   Remember that this class must implement the `Figure` interface, so it must provide the code for the complete set of methods that have been defined in such interface. These methods should be implemented with the available information in the class, that is: `getTag` and `printDescription` methods. It is not necessary to include those methods which are impossible to be programmed in the class (Java assumes that they are implicitly abstract), but subclasses are responsible to provide their code implementations.

The rectangle.

The next class to implement represents a rectangle and, obviously, its name will be `Rectangle`. This class inherits from `GeometricFigure` and implements the `Figure` interface. A rectangle is defined by two dimensions, the base and the height and both are assumed to be integer values.

1. Write the class declaration and its correspondings attributes.
2. Write the constructor and the basic accesor `get` and `set` methods.
3. Write the following methods of the class:

   ```
   public String getFigureType();

   public double area();

   public void drawTxt();
   ```

   The `drawTxt()` method must print the figure on the console. For example, a rectangle of base 6 and height 3 can be printed as follows:

   ```
   ******
   ******
   ******
   ```

4. Download the following code FiguresTesting and implement the code of the `main` method to create, first, an instance of `Rectangle` class and then print it on the console along with its description.
5. Answer the following questions:
   - Show the difference between a class and an object.
   - Which steps are involved in the instantiation's process of an object?
   - How is an object instantiated in Java?

REMEMBER: If a class implements an interface, automatically, all subclasses of it also implement that interface, even though this is not explicitly mentioned in its declaration. Notice that `Rectangle` implements the `Figure` interface though you haven't mentioned it in its declaration. That is because `Rectangle` inherits from a class that implements `Figure`.

The square.

A square is a type of rectangle where both base and height are equals. In Java, we can easily create a `Square` class by deriving it from `Rectangle` class.

1. Write the `Square` class so that it inherits from the `Rectangle` class of the previous section. `Square` class just only needs a constructor which receives a single input parameter: the side value. The constructor will invoke the constructor of the `Rectangle` superclass with the same value for the parameters of base and height
2. Test the new class by adding the necessary code to `FiguresTesting` class.
3. Notice that, thanks to inheritance, it is possible to invoke methods of the `Rectangle` superclass on an object of a derived type `Square` without the need of programming them again.
4. Answer the following questions:
   - What is inheritance?
   - How do you express in Java that one class *inherits* from another?
   - Which methods of the superclass are visible from the subclasses?
   - What is the meaning of method overriding?
   - Remember that, opposite to the rest of the methods of a class, subclasses don't automatically inherit constructors from the superclass, but they can be invoked by the use of `super()` keyword.

Reference to interfaces.

In this section, you must improve `FiguresTesting` class to provide a user interface in text mode that allows the user to choose the figure that he/she wants to print on the console, then asks for the correct parameters for each figure, creates an instance of the correct class and, finally, prints the figure description and the figure itself on the console.

Use the following menu:

```
1.- Create rectangle
2.- Create square
3.- Display figure
0.- Exit
```

If the user selects one of the 2 first options, the program must ask him/her for the correct data. If he/she selects option 3, the last figure will be printed on the console (including its description). The menu must be shown until the user select the option Exit.

To simplify the generation of the correct object from the data entered by the user, it would be useful to add a `readFigureData()` method to each class. This method would receive, as an input parameter, an object of type `BufferedReader` from which, it will read the data introduced by the user. Then, it would return a correct instantiated object of the class with the entered data. Declare it as a static method.

Answer the following questions:

- Which type are both instantiated objects (in options 1 and 2)?
- Which type is the variable that references them?
- Which methods from superclass are visible from the subclass?
- Can you use the same variable as a reference for diferent types of figures? Why?

1.3. Inheritance and references to the base class

30 min 30 Send

Objective

The resolution of this exercise expects that the student learns the basics of references to a base class in inheritance hierarchies.

Exercise

Download the following code listings (Position.java, Figure.java, Castle.java, Queen.java) and analyse both the class hierarchy and the `Position` class that is going to be used.

We are going to deal with a set of figure elements. As `Figure` class is abstract, we can't create instances from it, therefore, the collection of elements will be made up of objects from the `Castle` and `Queen` classes, indistinctly.

Implement the necessary code to hold, in the same collection and indistinctly, 2 objects of `Queen` class and 4 objects of `Castle` class.

Once the previous code has been implemented, traverse the collection invoking `public void whoAmI()` method to test the correct operation of the base class references.

Further questions

Once this exercise has been finished, answer the following questions:

- Which methods can be really invoked on the collection elements?
- If `Castle` class has implemented `void castle()` method ("enrocar" in Spanish), could it be possible to invoke that method from a reference to the base class?

- What should we have to do in order to be able to use the previous `void castle()` method from an object of `Castle` class that is pointed by a reference to `Figure` class?
- What should we do to know exactly to which class belongs every object pointed by a reference to the base class?

2. Homework

165 min

2.1. Object Oriented Programming and a possible solution for a cypher II

30 min 30 Send

Objective

This exercise tries to achieve an overall view of how Object Oriented Programmig allows us to reuse code, whenever we use it correctly.

Exercise

Let's suppose that we have a little knowledge about cypher theory with strings of characters. Let's say that we have heard of several cypher algorithms, most of them which are very difficult to implement. However, there is one called *Cesar*, which is based on character's rotation, has a reasonable implementation.

Cesar's algorithm is based on the order of the alphabet characters. Each character of the input text string is replaced by an output character that is calculated by adding N positions on the alphabet. For example if we find the 'A' character and the value of N is 3, the output character that will replace it is 'D'.

Implement an Object Oriented Programming solution that allows to change this algorithm easily by another one which could be better in the future. Use an `interface` in this solution.

2.2. Points and Figures (II)

45 min 45 Send

The main objective of this exercise is to review the use of inheritance in Java and its application to arrays of objects. Inheritance is based on the use of `abstract` and `extends` reserved identifiers. We will write a generic class and several specific subclasses that derive from the first. All abstract methods will be implemented and other methods will be overwritten to show the use of polymorphism when working wiht objects from a class hierarchy. Finally, we will use an array of objects from the previous hierarchy that will allow us to practice with all inheritance concepts.

In this exercise, we will calculate the total area of a collection of figures that have been previously created. Starting from the class `Triangle`, we will create a generic `Figure` class that will allow us to make abstraction when working with any type of figures. Once the behaviour of the `Figure` class has been defined, we will derive our `Triangle` class from it and then we will adapt it in order that it can be considered as a `Figure`. We will also create the `Square` class that will inherit also from `Figure`.

Once all figures are created, we will implement a class that contains an array of figures(for example, a triangle and a square) and then we will calculate the total area of them using class polymorphism to calculate the specific area of each one of them.

Inheritance. The `Figure` and `Triangle` classes

The `Figure` class represents a generic figure that will materialize later on a specific one (`Triangle`, `Square`, etc...). In Java, all these concepts are represented by abstract classes that must be declared with the `abstract` reserved keyword.

Note

An abstract class is a class that declares one or more abstract methods whose implementation is done in their corresponding subclasses. Review the theory if you don't remember this concept.

The first task of the exercise is to define the following `Figure` abstract class which represents a geometric figure:

```
public abstract class Figure {

    /** Name of the figure */
    String name;

    /** Constructor of the figure with a name */
    public Figure(String name) {
    }

    /** Calculates the area of a figure */
    abstract public double area();

    /** Indicates if the figure is regular or not */
    abstract public boolean isRegular();

    /** Creates a vertex with the coordinates given in the command line */
    public Point readVertex( String string ) {
        return null;
    }

}
```

Start by downloading the following code `Figure.java`.

Note

Notice that two methods have been declared as abstract, therefore they must be implemented in the corresponding subclasses.

Constructors of the `Figure` class.

1. Write the constructor of the class. The default value for `name` is "figure".

   ## Question

   Does it make sense to implement the constructor of an abstract class? What for?

The subclass `Triangle`.

The `Triangle` class now inherits from `Figure` and you must modify it so that it reflects such changes. To do so, start downloading the code listing of the `Triangle` class from this link `Triangle.java` and make the following changes:

Note

You will need to access the code of the `Point` class to be able to compile the `Triangle` class. You can download it from the following link `Point.java`.

1. Declare the `Triangle` class as follows:

   ```
   public class Triangle extends Figure
   ```

2. Modify all constructors so that, at the first line of the code, all of them invoke to the base class constructor of the `Figure` class. This is made by using the reference `super`.
3. Implement the following methods:
   - The `area()` method that calculates the area of the triangle. The prototype of the method is as follows:

```
public double area() {
/* complete */
}
```

The triangle's area is calculated according to the following formula:

Area = (base X height) / 2

Hint 1

Use the `distance(Point anotherPoint)` method, of the `Point` class, with the vertexes of the triangle.

Hint 2

To calculate the height of any triangle it is necessary to apply some trigonometry: assume that the triangle is *isosceles* or *equilateral* and suppose that the projection of the opposite vertex is on the middle point of the base.

- The `isRegular()` method that allows to know whether a triangle is regular or not. The prototype of the method is as follows:

```
public boolean isRegular() {
/* complete */
}
```

The method must return `true` if the length of all sides of the triangle are equal.

Hint

You need only to compare the length of the three sides of the triangle.

- The `toString()` method that allows to obtain a text string representation of the object. The prototype of the method is as follows:

```
public String toString() {
/* complete */
}
```

The method must return a string of characters with the values of the triangle according to some format. The output format of the text string is shown below (data values shown are only examples) :

```
TRIANGLE [NAME=triangle1] [NON REGULAR] : VERTEXES (1.0, 1.0),(3.0, 1.0),(2.0, 2.0)
```

Please, remember that what has been shown above is a formatted text string with example values for attributes. They are only used to show you how the output format is, so in the implementation of the method you must not use these values, instead of you have to work with attributes directly.

Hint 1

Use the `toString()` method of the vertexes of the triangle.

Hint 2

Remember that "+" operator, applied to text strings, allows to concatenate them.

The `main` method of the `Triangle` class

Now you must create a method to test the previous code. Create a `main` method in the `Triangle` class that carries out the following tasks:

1. Create three points.
2. Create a triangle from those previous points.
3. Display on the console a descriptive text string with the values of the triangle.

4. Finally, print the value of the triangle's area.

Inheritance. The `Square` class.

The `Square` class inherits also from `Figure` and has the following implementation:

```
public class Square extends Figure {

    /** Square vertexes */
    private Point vertex1;
    private Point vertex2;
    private Point vertex3;
    private Point vertex4;

    /** Constructor with a name - vertexes are read by command line */
    public Square(String name) {
    }

    /** Constructor with name and vertexes */
    public Square(String name, Point diagonalVertex1, Point diagonalVertex3) {
    }

    /** Private method to calculate the vertexes for the other diagonal*/
    private void otherDiagonal(Point vertex1, Point vertex3) {
    }

    /** Method implementation to calculate the area */
    public double area() {
      return 0;
    }

    /** Implementation of the abstract method to calculate if the figure is regular. */
    public boolean isRegular() {
      return false;
    }

    /** Returns a representative string of the square. */
    public String toString() {
      return null;
    }
}
```

Note 1

Notice that a square can be created from one of its diagonals, therefore, the constructor receives both vertexes of the diagonal as input parameters. To calculate the other two remaining vertexes, you have to implement another private method called `otherDiagonal(Point vertex1, Point vertex3)` that calculates and creates the other two vertexes of the second square's diagonal from those you have passed as parameters to the method, that is, it must calculate and create both vertexes `vertex2` and `vertex4` of the square.

Note 2

Also notice that, as it derives from the `Figure` abstract class, the `Square` class is forced to implement ALL abstract methods from `Figure`... Which are those?

Constructors of the `Square` class.

Download the skeleton of the `Square` class from this link Square.java and carry out the following tasks to implement the constructor of the class.

1. Firstly, implement the following private method that allows to create the vertexes of the second diagonal from the two vertexes of the first diagonal. The prototype of the method is as follows:

   ```
   private void otherDiagonal(Point vertex1, Point vertex3) {
   /* complete */
   }
   ```

   This method creates both two vertexes of the second square's diagonal (`vertex2` and `vertex4`) on the basis of the coordinates of the vertexes of the first diagonal that are passed as input parameters.

   ## Hint

Before creating those two points, you have to check up whether the first diagonal represented by `vertex1` and `vertex3` is vertical or horizontal since the calculation of the rest of the vertexes is easier in that case.

Alternatively, if you estimate propperly, this method can also be implemented according to the following steps:

1. Calculate the middle point (P) of the diagonal: P = ((x1+x2/2) , (y1,y2/2))
2. Calculate the director vector (Vector1) P->V1 (from the middle point to an extreme)
3. Extract the orthogonal director vector to the previous one and with the same modulus (Vector2)
4. Calculate the vertex vertice2 as: vertice2 = P + Vector2;
5. Calculate the vertex vertice4 as: vertice4 = P - Vector2;

2. Implement the constructor of the class.

Hint

Use the method that you have implemented before to calculate the square's vertexes.

Auxiliary methods of the `Square` class.

Implement the methods of the `Square` class that are specified next:

1. Write the `area()` method that allows to calculate the area of a square. The prototype of the method is as follows:

```
public double area() {
/* complete */
}
```

The method calculates the area of a square according to the following formula: Area = base X altura

Hint

Use the `distance(Point anotherPoint)` method of the `Point` class with the vertexes of the square.

2. Write the `isRegular()` method that allows to calculate whether a square is a regular figure or not. The prototype of the method is as follows:

```
public boolean isRegular() {
/* complete */
}
```

This method is obvious.

3. Write the `toString()` method that allows to obtain a text representation of the object. The prototype of the method is as follows:

```
public String toString() {
/* complete */
}
```

The method must return a string of characters with the values of the square according to some kind of format. In this case, the output format of the text string is shown below (data values shown are only examples) :

```
SQUARE [NAME=square1] : VERTEXES (3.0, 3.0),(5.0, 3.0),(5.0, 5.0),(3.0, 5.0)
```

Hint 1

You can reuse most of the code of the `toString()` method of the triangle.

Hint 2

Remember that "+" operator, applied to text strings, allows to concatenate them.

The `main` method of the `Square` class

Now you must create a method to test all the previous code. Create the `main` method in the `Square` class that must do the next tasks:

1. Create two points that will be the diagonal of your figure.
2. Create a square from those previous points.
3. Display on the console a descriptive text string with the values of the square.
4. Finally, print the value of the square's area.

Arrays. Calculation of the area of several figures.

The `FiguresArea` class, which contains an array of figures, is the one that will allows us to calculate the total area of a collection of figures. It has the following definition:

```
public class FiguresArea {

    /** The array of figures */
    Figure figures[] = null;

    /**  Constructor of the class with a fixed number of figures. */
    public FiguresArea(int figuresNumber) {
    }

    /** Calculates the total area of the array figures. */
    public double totalArea() {
        return 0.0;
    }

    /** Adds a new figure in the first empty position of the figures array. */
    public void addFigure (Figure f){
    }

    /** Prints a list with the array figures. */
    public void print() {
    }

    /** Main Program */
    public static void main(String args[]) throws Exception {
    }
}
```

The constructor of the class initialises the array with the maximum number of figures. The `addFigure()` method allows to add an object of the `Figure` class at the first free position of the array. The `totalArea()` method will add the respective areas of the figures that are contained in the array and will return the total sum of them. The `print()` method must print that total area and a descriptive message for every figure on the console.

Constructor.

Download the `FiguresArea` class from the following link FiguresArea.java and carry out the following task:

1. Implement the constructor of the class that initialises the array of figures with the maximum number of them.

Auxiliary methods of the `FiguresArea` class.

Implement the methods of the `FiguresArea` class that are specified next:

1. Write the `addFigure (Figure f)` method that allows to add an object `Figure` to the array. The prototype of the method is as follows:

```
public void addFigure (Figure f) {
/* complete */
}
```

The method must add a new object `Figure` to the figure's array.

Remember

According to the theory, `Figure` is *abstract* and, therefore there could not exist any object instances from that class. Object instances are, in fact, from the subclasses of `Figure`, that is: `Triangle, Square`.

2. Write the `totalArea()` method that allows to calculate the total area of the figures of the array.The prototype of the method is as follows:

```
public double totalArea() {
/* complete */
}
```

The method must calculate the total area of the figures that are contained in the array of figures. To do so, it crosses the array adding the corresponding area of each figure of the array.

Hint

Remember what it the use of abstract methods of the `Figure` class when implementing the method.

3. Write the `print()` method that allows to print the total area of the figures. The prototype of the method is as follows:

```
public void print() {
/* complete */
}
```

The method prints on console the result of the calculation of the total area of the figures. Use the previous method to achieve it.

Note

You can print also what figure is and to do so, use the `toString()` method of each figure.

Remember

Polymorphism is a consequence of applying inheritance to classes and it allows what is known as method *overwriting* by which, the same method can be implemented either at the base class or the subclass. At runtime, depending on the type of the object, Java's interpreter will invoke to the one of the correspondig class.

The `main` method of the `FiguresArea` class

Create the main program that has to carry out the following tasks:

1. Create an object of the `FiguresArea` class with two figures as maximum.
2. Create a figure of the `Triangle` class. To do so, you will have to create previously the 3 points that constitute the vertexes of the triangle and then pass them to the constructor of `Triangle`.
3. Create a figure of the `Square` class. To do so, you will have to create previously the 2 points that constitute diagonal of the square and then pass them to the constructor of `Square`.
4. Add the triangle and the square to the array of figures of the `FiguresArea` class. Use the method you have implemented previously to do it.
5. Print the result of the calculation of the total area of the figures. To do so, invoke to the corresponding method of the class.