

---

# PRACTICAL WORK OF LANGUAGES, TECHNOLOGIES, AND PARADIGMS OF PROGRAMMING

## PART I PROGRAMMING IN JAVA



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

---

Practical work 1

### Inheritance and polymorphism in Java

#### Contents

<b>1</b>	<b>Objectives of this practical session</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
2.1	Reviewing Java . . . . .	2
2.2	Using the type <code>Object</code> . Polymorphic Variables . . . . .	4
<b>3</b>	<b>Inheritance in Java</b>	<b>6</b>
<b>4</b>	<b>Abstract classes and polymorphism</b>	<b>9</b>
<b>5</b>	<b>Multiple Inheritance in Java</b>	<b>12</b>
<b>6</b>	<b>Evaluation</b>	<b>14</b>
<b>7</b>	<b>Appendix. Packages and visibility</b>	<b>14</b>

## 1 Objectives of this practical session

The **objective** of this practical session is to put in practice the concepts studied in the theoretical sessions about inheritance, polymorphism, and genericity in the Java programming language. This part consist of three sessions split into two parts: this first part is divided into two sessions and is devoted to the fundamentals of inheritance and polymorphism in Java. The second part comprises just one session and addresses genericity in Java. Practices will be developed using the BlueJ IDE (<http://www.bluej.org>).

In this first practical session, a concrete example is used to present the different characteristics of *inheritance* and *polymorphism*. The **problem** to be solved is the following one: *to design a Java class to store a set of geometric shapes that contains circles and triangles, and to solve it in such a way that the inclusion of new types of shapes only requires a minimal change in the designed software*. You will have to solve this problem in four different ways. In Section 2, you will solve it in two ways: first without resorting to inheritance or to polymorphism and later without inheritance but using polymorphic variables of type `Object`, predefined in Java. In Section 3, you will solve this problem by means of simple inheritance and polymorphic variables of type `Figura` defined by the programmer. Section 4 proposes to approach this problem using abstract classes with polymorphism in method definitions. The proposed problem is extended at the end of this section by including a new type of figure: cylinders. This extension is explained in Section 5 using multiple inheritance.

Several **exercises** are proposed throughout this practice bulletin. You should try to make them and you can pose questions to your practices teacher. Moreover, your practices teacher can extend the problems if required.

**Note:** The java code and, particularly, the name of classes, methods and variables will be kept in Spanish because we will try to maintain a unique proposed solution for all LTP groups, including the English based ARA groups and the Valencià groups.

## 2 Introduction

### 2.1 Reviewing Java

In this practice you will define some classes reusing other ones from the point of view of software reuse. Several implementation options are proposed leading to solutions more stable facing possible modifications of the problem specification. Moreover, some language features to reuse data structures and methods are suggested. They make use of new data types.

We are asked to design a Java class to store information about two kinds of geometric shapes: circles and triangles. In order to solve this problem,

it is possible to define a class for each different shape. The class `Circulo` is defined with two attributes `x` and `y` in order to specify where is located the shape in a two-dimensional space. An additional attribute `r` stores the radius of the circle. The three attributes are real numbers with `double` precision. The class `Triangulo` is defined with two coordinates, as with the circle, together with `base` and `altura` attributes, all of them of type `double` as well. These definitions are shown in Figure 1 together with a constructor method and the `toString()` method.

<pre>public class Circulo{     private double x,y;     private double r;      Circulo(double a, double b,             double c)         {x=a; y=b; r=c;}      public String toString ()     {return "Circulo:\n\t"+         "Posición: (" +x+", "+y+         ")\n\tRadio: "+r;}}</pre>	<pre>public class Triangulo{     private double x,y;     private double base,altura;      Triangulo(double cx,double cy,             double b, double a)         {x= cx; y = cy;         base = b; altura = a;}      public String toString()     {return "Triángulo:\n\t"+         "Posición: (" +x+", "+y+         ")\n\tBase: "+base+         "\n\tAltura: "+altura;}}</pre>
--	---

Figure 1: Definition of `Circulo` and `Triangulo` classes.

Now, we plan to design a class `GrupoFiguras` to store both types of shape using arrays, as illustrated in Figure 2.

```
public class GrupoFiguras {
    static final int MAX_NUM_FIGURAS = 10;
    private Circulo [] listaC = new Circulo [MAX_NUM_FIGURAS/2];
    private Triangulo [] listaT = new Triangulo [MAX_NUM_FIGURAS/2];
    private int numC=0, numT=0;
    public void anyadeCirculo(Circulo c) {listaC[numC++]= c;}
    public void anyadeTriangulo(Triangulo t) {listaT[numT++]= t;}
    public String toString(){
        String s= "Círculos:\n";
        for(int i = 0;i < numC; i++) s+="\n"+listaC[i].toString();
        s += "Triángulos:\n";
        for(int i = 0;i < numT; i++) s+="\n"+listaT[i].toString();
        return s;} }
```

Figure 2: Definition of the class `GrupoFiguras`

Since these structures are homogeneous, an array of circles and another one of triangles have to be defined. Both arrays will have a maximum capacity of `MAX_NUM_SHAPES/2`. The number of elements of each shape is represented by means of `numC` and `numT` attributes. These attributes are incre-

mented after each insertion performed by methods `anyadeCirculo(Circulo)` and `anyadeTriangulo(Triangulo)`. The `toString` method traverses the shapes stored in each array in order to show the information associated to each element by using the `toString` method of this element.

A group of figures is created in the `UsoDeGrupoFiguras` class illustrated in Figure 3. One circle and one triangle are added to this group of shapes and the information is printed to standard output afterwards.

```
public class UsoDeGrupoFiguras{
    public static void main (String args[]){
        GrupoFiguras g = new GrupoFiguras();
        g.anyadeCirculo(new Circulo(10,5,3.5));
        g.anyadeTriangulo(new Triangulo(10,5,6.5,32));
        System.out.println(g);}
}
```

Figure 3: Definition of `UsoDeGrupoFiguras` class.

The execution of `UsoDeGrupoFiguras` produces the following output:

```
Standard Output
Círculos:
Círculo:
    Posición: (10.0,5.0)
    Radio: 3.5
Triángulos:
Triángulo:
    Posición: (10.0,5.0)
    Base: 6.5
    Altura: 32.0
```

## 2.2 Using the type `Object`. Polymorphic Variables

In the example of the previous section, two arrays were used due to the impossibility of storing references of two different types in the same array. This problem is typically solved by using the `Object` class, predefined in Java. All other object types (both those predefined in the language and those defined by the user) inherits from or are extensions of `Object`. In practice, this implies that variables of this type can contain references to instances of any other object type, which leads to a polymorphism of variables.

Let us see how to use this type to reduce the number of arrays in the example of Figure 4. This example is a redefinition of the class `GrupoFiguras` which defines only one array `listaFiguras` of type `Object` (line 3). This allows us to store any object in the components of an array no matter its type, including objects of type `Circulo` and `Triangulo`, as can be observed in the assignments of variables `c` and `t` in methods `anyadirCirculo` and `anyadirTriangulo` (lines 5 and 6). These two methods are the only public

ones that can alter the data structure. Note that they only allow circles and triangles in their formal parameters. If we try to add an object of type `Object` to a group `g` of shapes with the following instruction

```
g.anyadeTriangulo(new Object());
```

the compiler would report an error, since the type `Triangulo` is an `Object` but not every `Object` is a `Triangulo`. If we try the same with any other type different from the parameters, we will get the same behavior.

**Exercise 1** *Implement the classes `Circulo`, `Triangulo`, `GrupoFiguras` and `UsoDeGrupoFiguras`. In the latter class, add some objects of both types, compile and observe the results. You have to implement `GrupoFiguras` class using whenever possible the `Object` class.*

```
1 public class GrupoFiguras{
2     static final int MAX_NUM_FIGURAS = 10;
3     private Object [] listaFiguras = new Object [MAX_NUM_FIGURAS];
4     private int numF=0;
5     public void anyadeCirculo(Circulo c) {listaFiguras[numF++]= c;}
6     public void anyadeTriangulo(Triangulo t) {listaFiguras[numF++]= t;}
7     public String toString(){
8         String s= "Círculos:";
9         for(int i = 0;i < numF; i++)
10             if (listaFiguras[i] instanceof Circulo) s+="\n"+listaFiguras[i];
11         s+= "\nTriángulos:";
12         for(int i = 0;i < numF; i++)
13             if (listaFiguras[i] instanceof Triangulo)s+="\n"+listaFiguras[i];
14         return s;}}
```

Figure 4: Class `GrupoFiguras` using the polymorphic type `Object`

In order to maintain the behavior of the `toString` method (grouped by the type of the shapes), we need to distinguish between the two types of each `Object` that is stored in the array. Fortunately, it is possible, in Java, to determine whether an object is an instance of a concrete class or not by using the instruction `instanceof` with the following syntax:

```
variableReferenceToObject instanceof nameOfTheClass
```

An example of the use of `instanceof` can be found in lines 10 and 13 of Figure 4, where this instruction is used to select the objects of a given shape.

The use of `Object` as a generic type forces the programmer to constantly ask about the type of the instances and to perform type casts, which means that the responsibility of a correct use depends on the programmer. This would happen, for example, if we try to override the `equals` method of class `Circulo`, a method which is inherited from `Object`:

```
public boolean equals (Object c){
    return this.r==((Circulo)c).r;}

```

where the type of `c` is cast to `Circulo` in order to notify the compiler that `c` is referencing an object with an attribute `r`. This approach is not correct since it is assumed that the formal parameter `c` is a reference to a circle, which is not necessarily the case.

**Exercise 2** *Overwrite the method `equals(Object)` of classes `Circulo`, of class `Triangulo` and of class `GrupoFiguras`. It will be assumed that two shapes are equal when they have the same type and their attributes are the same. A group of shapes is equal to another group if they contain the same shapes no matter the order in which they appear and no matter the number of times they appear. Try the implemented methods `equals(Object)` in `UsoDeGrupoFiguras` by comparing different instances. What happens when you compare objects of different shape? Consider which changes are required in `anyadeCirculo`, `anyadeTriangulo` and `equals` methods of the `GrupoFiguras` class if the groups of shapes were sets (that is, if they could not contain repeated elements).*

### 3 Inheritance in Java

Continuing with the example of shapes, and observing that both circles and triangles are located in a position, we can consider the position as a common factor of a new class `Figura` as illustrated in Figure 5. The data structure of objects that are instances of this class only contains two attributes `x` and `y` of type `double`, a constructor method, and the `toString` method.

```
public class Figura{
    private double x, y; //Posición de la figura
    public Figura (double x, double y)
    {this.x = x; this.y = y;}
    public String toString()
    {return "Posición: (" + x + ", " + y + ")";}
}

```

Figure 5: Class `Figura`.

Now, we can extend the class `Figura` with the class `Circulo`:

```
public class Circulo extends Figura{
    private double r; ...}

```

Since the attributes of class `Figura` are private, they are not visible to other classes, including derived ones. If we want to define the constructor of a circle using the parameters `x` and `y`, related with position, and the parameter of the radius `r`, we should call the constructor method of the parent class. The

word **super** is used to refer to the parent class. The constructor method will be as follows:

```
1 public Circulo (double x, double y, double r){
2     super(x,y);
3     this.r=r;
4 }
```

where the constructor method of the parent class (**Figura**) is called in line 2. Let us note that the word **super** is used instead of the name of the parent class. An important restriction when using the constructor of the parent class is that this call has to be the first instruction in the body of the method. Moreover, when a method inherited from the parent class is rewritten, the method from the parent can still be used in the derived class by using the dot notation as in the following example: **super.toString()**;

It is possible to relax the visibility of attributes and methods of a parent class to make them visible by all derived classes (including those which belong to a different package) using the **protected** modifier. If we apply it to the class **Figura**, we can leave only its default constructor method, since all the attributes will be visible by the class **Circulo**. The class **Figura** would be as follows:

```
public class Figura{
    protected double x, y; //Posición de la figura
    public String toString(){
        return "Posición: (" + x + ", " + y + ")"; }
}
```

Figures 6 and 7 illustrate the use of inheritance by extending the class **Figura**. As can be observed in lines 7 and 8, the call to the parent constructor (**super(x,y);**) has been replaced by the direct use of the inherited attributes **x** and **y**, since they are now accessible in the class **Circulo**.

Let us also observe that **super.x** is used in line 7 but **this.y** is used in line 8, **super** and **this** are equivalent in this context because these attributes are also part of **Circulo**. This difference disappears in lines 23 and 24. Note also that the call to the **toString** method of **Figura** is kept in line 14. Although this call is not strictly necessary (as you can observe in line 30), it is better to reuse the method of the parent class in order to make the code more maintainable as in the following example: if someone decides to change word **Posición** by **Punto**, the change in the parent class is automatically propagated to derived classes.

The class **GrupoFiguras**, redefined in Figure 8, uses the type **Figura** in the array instead of the type **Object**. This implies that each array component can only reference objects of this type and its derived classes (**Circulo** and **Triangulo**). In this way, we do not have to worry about the possibility of array components referencing other types not related to **Figura**. Even so,

<pre> 1 public class Circulo 2     extends Figura{ 3 private double r; 4 5 public Circulo(double x,double y, 6     double radio){ 7     super.x = x; 8     this.y = y; 9     r= radio;} 10 11 12 public String toString(){ 13     return "Círculo:\n\t"+ 14     super.toString()+"\n\tRadio: "+r; 15 } 16 } </pre>	<pre> 17 public class Triangulo 18     extends Figura{ 19 private double base, altura; 20 21 public Triangulo(double x,double y, 22     double base, double altura){ 23     this.x = x; 24     this.y = y; 25     this.base= base; 26     this.altura=altura;} 27 28 public String toString (){ 29     return "Triángulo:"+ 30     "\n\tPosición("+x+", "+y+ 31     ")\n\tBase: "+base+ 32     "\n\tAltura: "+altura;}} </pre>
--	--

Figure 6: Class `Circulo`

Figure 7: Class `Triangulo`

it would be possible to reference objects of type `Figura` created by means of the default constructor. As we would see in the next section, this can be avoided by transforming the class `Figura` into an abstract class. A unique modifier method `anyadeFigura` is required in this class. The `toString` method is simplified with respect to the previous design since now it is not required to group together objects of the same shape. The new definition of `toString` method changes the output of `UsoDeGrupoFiguras` class as follows:

Standard Output

```

Círculo:
  Posición: (10.0, 5.0)
  Radio: 3.5
Triángulo:
  Posición(10.0,5.0)
  Base: 6.5
  Altura: 32.0

```

The classes described before are depicted in Figure 9 using boxes and the relations between them are illustrated using different types of lines and arrows. There are two kinds of relations between classes: On 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 than the derived one. On the other hand, the relation *USES-A* is represented by a dotted line. You can observe in Figure 9 a diagram where the class `GrupoFiguras` makes use of `Figura`, `Circulo` and `Triangulo`, but `GrupoFiguras` is not derived from other class. The classes `Circulo` and `Triangulo` do not use any data type,



```

public class GrupoFiguras{
    static final int MAX_NUM_FIGURAS = 10;
    private Figura [] listaFiguras = new Figura [MAX_NUM_FIGURAS];
    private int numF=0;
    public void anyadeFigura(Figura f) {listaFiguras[numF++]=f;}
    public String toString(){
        String s= "";
        for(int i = 0;i < numF; i++) s+="\n"+listaFiguras[i];
        return s;}}

```

Figure 8: Class GrupoFiguras using the type Figura

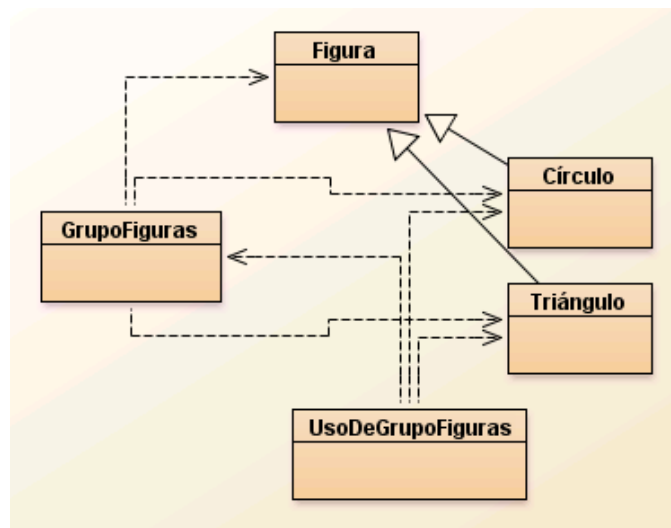


Figure 9: *IS-A* and *USES-A* relationships.

just inherit from the type `Figura`. Finally, the class `UsoDeGrupoFiguras` uses the classes `Circulo`, `Triángulo` and `GrupoFiguras` but it does not use the class `Figura`.

**Exercise 3** Define a class `Rectangulo` that inherits from `Figura`. A rectangle is a shape with two attributes of type `double` (base and altura) and with the same methods as `Circulo` and `Triángulo`. Is there any change in `GrupoFiguras`? Add a rectangle in the group of figures defined by the class `UsoDeGrupoFiguras` to check it works.

## 4 Abstract classes and polymorphism

We have seen in previous section how to restrict the types of objects that can be included in the array `listaFiguras`. Only objects of type `Figura`

and its derived ones are allowed, but this also implies that an instance of **Figura** can be created and stored in the array:

```
listaFiguras[posición] = new Figura();
```

However, the initial objective of this session was to represent a group of circles and triangles, not objects of type **Figura** which only contain a position. It is possible to avoid this problem by allowing the existence of objects of type **Circulo** or **Triangulo** but not of type **Figura**. This is achieved by defining the last class as an abstract class using the reserved word **abstract**:

```
public abstract class Figura{  
    protected int x, y; //Posición de la figura  
    ... }  

```

Observe that the modifier of attributes **x** and **y** is changed to **protected** in order to make them visible to the derived classes.

**Exercise 4** *Another way to avoid the inclusion of instances of the class **Figura** consists of checking the type of object received as parameter by the **anyadeFigura(Figura)** method and to insert only those objects extending **Figura**. What should be done each time a new type of **Figura** is included into the application? Does inheritance offer some advantage in terms of application maintenance?*

Abstract classes are usually used to create a hierarchy of classes that share some common behavior.

**Exercise 5** *Let us suppose we want a new method **area()**, available for any type of shape, that returns the area of the shape. Define an abstract method **area()** in the class **Figura** that returns a value of type **double**.*

Figure 10 shows the definitions of classes **Circulo** and **Triangulo** after the modification of class **Figura** with the inclusion of the method **area()**.

The hierarchy of classes **Circulo** and **Triangulo** is illustrated in Figure 11. We can observe in this figure that they are descendants of an abstract class **Figura**. The class **GrupoFiguras** uses the type **Figura** without changing its code. The test class **UsoDeGrupoFiguras** also remains unchanged and makes use of the classes **Circulo**, **Triangulo** and **GrupoFiguras** to create instances of these classes.

**Exercise 6** *After defining the method **area()** in class **Figura**, its derived classes should also implement this method. Implement it in all its derived classes, including the class **Rectangulo** that you have defined before. Check that your implementation works correctly by showing and checking the area of instances created in the class **UsoDeGrupoFiguras**.*

```

public class Circulo
    extends Figura{
private double r;

public Circulo(double x,double y,
    double radio){
    super.x = x;
    this.y = y;
    r= radio;}

public String toString(){
    return "Círculo:\n\t"+
    super.toString()+"\n\tRadio: "+r;
}
}

public class Triangulo
    extends Figura{
private double base, altura;

public Triangulo(double x,
    double y,double b,double a){
    this.x = x;
    this.y = y;
    this.base= b;
    this.altura=a;}

public String toString (){
    return "Triángulo:"+
    "\n\tPosición("+x+", "+y+
    ")\n\tBase: "+base+
    "\n\tAltura: "+altura;}}

```

Figure 10: Subclasses Circulo and Triangulo

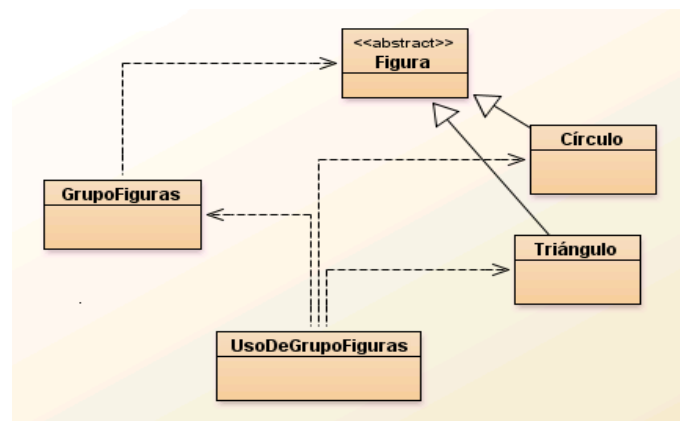


Figure 11: IS-A and USES-A relationships.

**Exercise 7** Define a method called `area()` in class `GrupoFiguras`, that computes the sum of all the areas of shapes contained in the group of shapes. To this end, traverse all shapes references in the `listaFiguras` 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.

Now, we can use inheritance to extend the hierarchy of classes defined so far. Let us derive a new class `Cilindro` from class `Circulo` as illustrated in Figure 12.

An implementation of class `Cilindro` is shown in Figure 13. You can observe a new attribute `a` of type `double` to represent the height of the

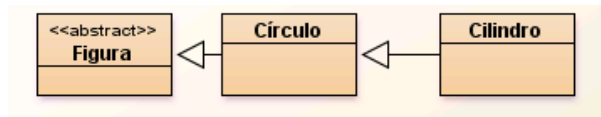


Figure 12: Diagram of the hierarchy of classes.

cylinder. Two constructors are defined in this class. The first one (lines 3-5), receives four parameters: the two position coordinates, the radius of the base, and the height. In its first instruction (line 4) **super** is used in order not to overwrite the assignments of values to the attributes of a circle. The second constructor (lines 6 and 7) receive two parameters: a circle **c** and a height. In the line 7 of this constructor, the constructor with four parameters (defined in lines 3-5) is used. Note that **this** is used instead of the name of the class. The attribute **r** of parameter **c** is accessible in this class since it is defined as **protected** in **Circulo**.

```

1 public class Cilindro extends Circulo{
2     protected double a;
3     Cilindro(double x, double y, double radio, double altura){
4         super(x,y,radio);
5         a= altura;}
6     Cilindro(Circulo c, double altura)
7         {this(c.x, c.y, c.r, altura);}
8     public double volum()
9         {return super.area()*a;}}
  
```

Figure 13: Class **Cilindro**

**Exercise 8** *Implement the method **area()** in the class **Cilindro**. This method should calculate the area of a cylinder by adding the area of a rectangle and twice the area of a circle. You have to create an instance of **Rectangulo** of width the perimeter of the circle of the cylinder (computed from the radius) and the same height as the cylinder. The area of the circle has to be computed by using the method **area()** over **super**.*

## 5 Multiple Inheritance in Java

Let us suppose that we implement an interface **Volumen** as follows:

```

public interface Volumen{
    public double volumen();
    public double superficie();}
  
```

and let us also suppose that class **Cilindro** is redefined using this interface:

```

public class Cilindro extends Circulo implements Volumen{
    protected double a;
    Cilindro(double x, double y, double radio, double altura){
        super(x,y,radio);
        a= altura;}
    Cilindro(Circulo c, double altura)
        {this(c.x, c.y, c.r,altura);}
    public double volumen()
        {return super.area()*a;}}

```

When we try to compile it, the following error is reported:

```

Cylinder is not abstract and does not override abstract method
surface() in Volume

```

indicating that there are two possibilities: either to declare the new class as an abstract class or to implement the method `superficie()`. Moreover, since the method `volumen()` was already implemented, this does not generate any error.

**Exercise 9** *Why do we get this error message? Make the necessary changes to solve it.*

Once the whole interface is implemented, the diagram of the hierarchy is as depicted in Figure 14 where you can observe that the implementation of the interface `Volumen` by the class `Cilindro` is represented by a dotted line with a solid arrow.

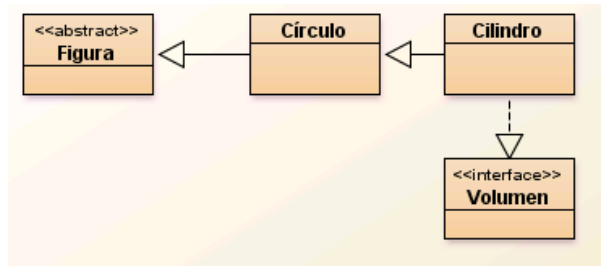


Figure 14: Diagram of the hierarchy of classes and interfaces

Defining variables of an interface type allows those variables to reference objects of any class implementing the functionality described by this interface. In this way, we can be sure that these objects have the methods that are specified in the interface. The following example makes use of method `volumen()`

```

Volumen figuraConVolumen = new Cilindro(new Circulo(10,5,4.5),10.1);
System.out.println(figuraConVolumen.volumen());

```

Interfaces allows classes to offer a common functionality despite being defined in different class hierarchies. This means a use of inheritance different from that of the usual class hierarchy.

**Exercise 10** *Add to the class `GrupoFiguras` a method `volumen()` that computes the sum of the volumes of all the shapes contained in the group that also implement the interface `Volumen`. You should traverse the figures referenced in the `listaFiguras[]` attribute in order to check whether or not they implement the interface `Volumen`. This check can be easily done by means of the `instanceof` instruction.*

## 6 Evaluation

The attendance to practical sessions is mandatory to pass this course. The evaluation of the first part of the practical work on Java will be done by means of an individual exam.

## 7 Appendix. Packages and visibility

When creating large programs and/or working in group, it is wise to divide the code into parts or packages in order to reduce the compile time (facing source code updates), to increase the efficiency of the group work and also to control the access to classes and interfaces avoiding clashes due to the use of the same identifiers.

*Packages* are directories containing `.class` files with pre-compiled classes. They follow a hierarchy different from that of inheritance. Usually, packages include classes following a criteria related with their functional cohesion. For instance, the package `java.awt.geom` contains classes to define operations defined on objects related with two-dimensional geometry.

Classes can be imported individually or at the package level by including the `*` wild card at the end of the instruction in order to indicate the access to all the classes contained in the package. Imports are explicitly defined at the beginning of the file. For instance:

```
import java.applet.Applet;  
import java.awt.geom.*;
```

where dots separate subpackages, as will be seen later. This notation provides us some control over namespaces as well, differentiating classes with the same name. An example in Java is the existence of three predefined classes with the name `Timer`, each one defined on a different package and with a different functionality. The three classes can be used in the same program by importing them at the beginning of the file:

```
import java.util.Timer
import javax.management.timer.Timer
import javax.swing.Timer
```

and by defining variables using the correct path to locate the directory where the class is defined:

```
java.util.Timer t1 = new java.util.Timer();
javax.management.timer.Timer t2 = new javax.management.timer.Timer();
javax.swing.Timer t3 = new javax.swing.Timer();
```

It is also possible to perform static imports in order to avoid the class names when using static methods, references to static constants and so on. For instance:

```
import static java.lang.Math.*;
...
double r = cos(PI * theta);
```

It is possible to define more than one class in the same file but only one of them can be public. In this case, the file-name (using the `.java` extension) must coincide with the public class. The visibility of the rest of the classes defined in the file is limited to the package itself. Packages usually contain just one class, although several classes are sometimes defined when they are only required inside the package. In order to use Java packages, it is required to generate a folder structure with the same hierarchy as libraries specified in the packages. In order to specify that a given class belongs to a given library, the name of the package must be specified in the first line of the file using the following syntax:

```
package package1.package2. ... .packageN;
```

The directory paths defined using the dot notation describe the relative path from the folder of the overall project to the folder where the class is defined.

For instance, let us suppose that we have a project in the directory `lineales`, and imagine also that we want to include in the package `modelos` the classes defined in file `Pila.java`, where `modelos` is a sub-package of `librerias`. We should write `package librerias.modelos;` in the first line of the file. This means that classes defined in `Pila.java` file are available in the `/.../lineales/librerias/modelos` folder.

There is another issue related with packages and with the use of some commands for compiling (`javac`), executing (`java`) and generating documentation (`javadoc`). By default, these commands search libraries from the folder where JDK is installed and from the folder where they are executed. When other folders are required, these commands rely in an environment variable called `CLASSPATH`. This variable defines a sequence of paths where packages are searched.

Since class accessibility not only depends on the package where they are defined, let us briefly review in a general way the visibility of classes and of their internal components depending on how these components are defined.

### Class definition

Syntax:

```
accessModifier classModifier class ClassName
                                     [extends ClassName]
                                     [implements interfaceList]
```

where

**accessModifier** indicates where this class can be accessed from.

**public** the class is visible from any other class no matter its package.

*without specification* the class is only accessible from classes defined in the same package.

**private** it is only visible from the class itself<sup>1</sup>.

**classModifier** affect derived classes.

**abstract** for abstract classes.

**final** prevent the class to be derived.

**Inheritance** which kind of class is derived.

**extends** is used to indicate that the class is derived from **ClassName**. Java is based on single inheritance and classes can only be derived from a sole parent class. When the extends clause is not used, it is assumed that the class is directly extending **java.lang.Object**

**implements** denotes that this class satisfies the interfaces from the list **interfaceList**, which may contain several interfaces separated by commas.

### Variable definition

Syntax:

```
[visibilityModifier] [attributeModifier] type variableName;
```

where

**visibilityModifier** delimit the access from outside the class.

**public** accessible from any class.

**private** can only be accessed from the class where it is defined.

**protected** accessible from classes from the same package as well as their derived classes even if they belong to a different package.

*without specification* accessible from any class from the same package.

**attributeModifier** special characteristics.

**static** the variable does not belong to any object but is rather common to all objects from the class.

**final** the value assigned to the variable cannot be changed.

---

<sup>1</sup>It is possible to define classes inside other classes. The former ones are known as *internal classes*.



**transient** the attribute is excluded from serialization even if the class implements the **Serializable** interface.

**volatile** means that this attribute can be accessed in an asynchronous way for several threads so that the compiler should not modify the execution order of instructions.

### Class method definition

Syntax:

[**visibilityModifier**] [**functionModifier**] type functionName (parameterList)  
where

**visibilityModifier** the same rules as for attributes.

**functionModifier** can have the following values:

**static** is used to specify class methods, which are not applied to objects. When a static method is called from another class, it must be preceded by the class name followed by a dot.

*without specification* is a dynamic method.

**final** the method cannot be overwritten in a derived class.

**abstract** the implementation is delegated to a derived class.

**native** is written in native code<sup>2</sup> resulting from a compilation process.

**synchronized** is executed in exclusively by a thread <sup>3</sup>

You can find more information relating packages and visibility in Java in the following url:

<http://docs.oracle.com/javase/tutorial/java/java00/accesscontrol.html>

---

<sup>2</sup>Native code is directly executable by the processor and can be obtained by compiling a high level language.

<sup>3</sup>When several concurrent threads are trying to execute at the same time a dynamic method over the same object, a method marked as synchronized is only executed by one thread and the others must wait until it finishes. If the method is static, only one static methods is executed at the same time.