

---

# PRÀCTIQUES DE LLENGUATGES, TECNOLOGIES I PARADIGMES DE PROGRAMACIÓ

## PART I PROGRAMACIÓ EN JAVA



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

---

### Pràctica I

Herència i Polimorfisme en Java

### Índex

<b>1</b>	<b>Objectiu i plantejament de la Pràctica</b>	<b>2</b>
<b>2</b>	<b>Introducció</b>	<b>2</b>
2.1	Repassant Java . . . . .	2
2.2	Emprant el tipus Object. Variables polimòrfiques . . . . .	4
<b>3</b>	<b>Herència en Java</b>	<b>6</b>
<b>4</b>	<b>Classes abstractes i polimorfisme</b>	<b>9</b>
<b>5</b>	<b>Herència múltiple en Java</b>	<b>12</b>
<b>6</b>	<b>Avaluació</b>	<b>14</b>
<b>7</b>	<b>Apèndix. Paquets i visibilitat</b>	<b>15</b>

## 1 Objectiu i plantejament de la Pràctica

L'objectiu d'aquesta primera part de pràctiques és que aprofundeixes en els coneixements estudiats en teoria sobre herència, polimorfisme i genericitat en el llenguatge Java. Aquesta part la realitzaràs en tres sessions dividides en dues pràctiques: La primera pràctica, de dues sessions, es dedica a aspectes bàsics d'herència i polimorfisme en Java, la segona, d'una sessió, està dedicada als tipus genèrics. Les pràctiques les realitzaràs usant l'entorn BlueJ (<http://www.bluej.org/tutorial/tutorial-spanish-1.pdf>).

En aquesta primera pràctica se te planteja la resolució d'un problema mitjançant el qual aniràs veient diversos aspectes de la *herència* i el *polimorfisme*. El **problema** és el següent: *dissenyar una classe per guardar un grup de figures geomètriques que continga només cercles i triangles, de manera que la incorporació de nous tipus de figures supose canvis mínims en el programari dissenyat*. El problema es resoldrà de quatre formes diferents. A la secció 2 es començarà fent-lo de dues maneres: primer sense utilitzar herència ni polimorfisme i després sense herència però amb variables polimòrfiques de tipus `Object`. A la secció 3 es resoldrà usant herència simple i variables polimòrfiques de tipus `Figura` definit per el programador. A la secció 4 l'hauràs de resoldre usant classes abstractes utilitzant polimorfisme de mètodes. Al final d'aquesta secció, s'ampliarà el problema a resoldre afegint un nou tipus de figura *cilindre*. Aquesta ampliació s'explica a la Secció 5 fent ús de herència múltiple.

Al llarg del butlletí de la pràctica es proposen diversos **exercicis** que has d'intentar realitzar plantejant els teus dubtes al teu professor de pràctiques. Així mateix, el teu professor de pràctiques pot ampliar el problema a resoldre en les sessions de pràctiques.

## 2 Introducció

### 2.1 Repassant Java

En aquesta pràctica definiràs classes reutilitzant altres classes des del punt de vista de la reutilització del programari. Es proposen diverses opcions de implementació avançant cap a solucions més estables davant de possibles modificacions del problema. A més, es plantegen algunes possibilitats que ofereix el llenguatge per reutilitzar les estructures de dades i mètodes fent ús de nous tipus de dades.

Se'ns demana dissenyar una classe per guardar informació sobre dos tipus de figures geomètriques: cercles i triangles. Per a resoldre'l podem definir una classe per a cada tipus de figura. La classe `Circulo` es defineix amb dos atributs `x` i `y` per al punt en què està situada la figura en un espai de dues dimensions, a més conté un atribut `r` que representa el radi del cercle. Els tres atributs són del tipus real `double`. La classe `Triangulo` es defineix, igual

que els cercles, amb dues coordenades però amb els atributs **base** i **altura** també de tipus **double**. A la Figura 1 s'il·lustren aquestes definicions i en ambdues classes es defineix un constructor i el mètode **toString()**.

<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 "Círculo:\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>
--	---

Figura 1: Definició de les classes **Circulo** i **Triangulo**

Ara ens plantegem dissenyar una classe **GrupoFiguras** on guardar figures d'ambdós tipus utilitzant arrays tal com apareix a la Figura 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;}}
```

Figura 2: Definició de la classe **GrupoFiguras**

Com que aquestes estructures són homogènies, definim un array de cercles i un altre de triangles, cada un amb una capacitat **MAX\_NUM\_FIGURAS/2**, com a màxim. La quantitat de figures de cada tipus queda representada mitjançant els atributs **numC** i **numT** respectivament. Aquests atributs s'incrementen amb cada inserció realitzada pels mètodes **anyadeCirculo(Circulo)** i **anyadeTriangulo(Triangulo)**. El mètode **toString()** recorre les figures referenciades en els arrays per a mostrar la informació de cada figura segons els seus propis mètodes **toString()**.

En la classe `UsoDeGrupoFiguras` de la Figura 3 es crea un grup de figures al qual se li afegeix un cercle i un triangle i s'escriu la informació del grup.

```
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);}}
```

Figura 3: Definició de la classe `UsoDeGrupoFiguras`

La seva execució produeix l'eixida:

Eixida Estàndar	
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 Emprant el tipus `Object`. Variables polimòrfiques

En l'exemple de la secció anterior es van fer servir dues arrays a causa de la impossibilitat d'emmagatzemar objectes de diferent tipus en un mateix array. Aquest problema pot solucionar-se fent servir la classe predefinida en Java `Object`, de la qual hereten tots els tipus definits en el llenguatge i els definits pel programador. És a dir, que tots els objectes són extensions d'aquest tipus. A la pràctica, això implica que les variables d'aquest tipus poden contenir objectes de qualsevol tipus, donant lloc a un polimorfisme de variables.

Vegem com emprar aquest tipus per a alleugerir la quantitat d'arrays en l'exemple de la Figura 4. Aquest exemple és una redefinició de la classe `GrupoFiguras` definint un únic *array* `listaFiguras` de tipus `Object` (línia 3). Això permet emmagatzemar en les components de l'*array* qualsevol tipus d'objecte, entre ells, els de tipus `Circulo` i `Triangulo`, com pot veure's en les assignacions de `c i t` en els mètodes `anyadirCirculo` i `anyadirTriangulo` (línies 5 i 6). Aquests dos mètodes són els únics públics que alteren l'estructura de dades i en els seus paràmetres formals només permeten cercles i triangles que seran les úniques figures guardades. Si intentàrem afegir a un grup de figures `g` un objecte de tipus `Object` amb una instrucció com

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

el compilador ens donaria un error, ja que un tipus `Triangulo` és `Object` però no tot `Object` és un `Triangulo`. En aquest estat de la solució, ocorreria el mateix si ho intentàrem amb qualsevol altre tipus diferent al dels paràmetres.

**Exercici 1** *Implementa les classes `Circulo`, `Triangulo`, `GrupoFiguras` i `UsoDeGrupoFiguras`. En aquesta darrera classe, afegeix diverses figures de tots dos tipus, compila, executa i observa el resultat. La classe `GrupoFiguras` hauràs de definir-la fent servir, en el possible, el tipus `Object`.*

```
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;}}
```

Figura 4: Classe `GrupoFiguras` emprant el tipus polimòrfic `Object`

Per a mantindre les figures agrupades pels seus tipus en el valor de tornada del mètode `toString()`, es necessita distingir entre els dos tipus de `Object` que es referencien en l'array `listaFiguras`. En Java és possible preguntar si una instància d'una classe és d'un tipus determinat mitjançant la instrucció `instanceof` emprant la sintaxi:

`variableObjecte instanceof NomDeLaClasse`

Així, en les línies 10 i 13 de la Figura 4 es pregunta pel tipus dels objectes per a seleccionar-los.

L'ús del tipus `Object` obliga al programador a preguntar constantment pels tipus de les instàncies i realitzar conversions forçades de tipus deixant l'ús correcte dels tipus en mans del programador. Això últim ocorreria, per exemple, si en la classe `Circulo` sobreescrivim el mètode `equals` heretat de la classe `Object`:

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

on el tipus de `c` es força al tipus `Circulo` perquè el compilador sàpia que `c` fa referència a un objecte que conté un atribut `r`. Ací es corre un alt risc

ja que es dona per descomptat que el paràmetre formal `c` va a contenir una referència a un cercle.

**Exercici 2** *Sobreescriu el mètode `equals(Object)` per a les classes `Circulo`, `Triangulo` i `GrupoFiguras`. Per a això suposarem que dues figures són iguals si contenen exactament els mateixos valors en els seus atributs, i que dos grups de figures són iguals si contenen les mateixes figures sense importar l'ordre ni la quantitat de vegades que apareguen. Prova els mètodes `equals(Object)` en `UsoDeGrupoFiguras` comparant objectes entre si. Què ocorre si compares figures de tipus diferents? Considera els canvis que realitzaries en els mètodes `anyadeCirculo`, `anyadeTriangulo` i `equals` de la classe `GrupoFiguras` si els grups de figures foren conjunts, és a dir, sense elements repetits.*

### 3 Herència en Java

Continuant amb l'exemple de les figures, i observant que tant els cercles com els triangles estan situats en una posició del plànol, es pot traure com a factor comú la posició d'ambdues i formar una nova classe **Figura** com apareix en la Figura 5. L'estructura de dades dels objectes d'aquesta classe només conté dos atributs `x` i `y` de tipus `double`, un mètode constructor i el mètode `toString()` per a obtenir la seua informació.

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

Figura 5: Classe Figura

Ara podem estendre la classe `Figura` amb la Classe `Circulo`:

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

Com els atributs de la classe `Figura` són privats, no són visibles en una altra classe incloses les seues derivades. Si es vol definir el constructor d'un cercle amb els dos paràmetres de la posició `x` i `y`, i el paràmetre del radie `r`, s'haurà de cridar al constructor de la classe base. Per a referenciar a la classe base s'usa la paraula reservada `super`. El constructor quedaria com:

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

on en la línia 2 es crida al constructor de la classe base (**Figura**). Fixa't en què només s'usa la paraula **super** i no el nom de la classe base. Una restricció important és que si s'invoca al constructor de la classe base, ha de ser la primera instrucció del cos del mètode. A més, si es sobreescriu algun mètode heretat, el de la classe base pot invocar-se en la derivada amb la notació punt, per exemple: **super.toString()**;

Es pot relaxar la visibilitat dels atributs i mètodes d'una classe base per que siguin visibles per totes les seves classes derivades (fins i tot quan pertanyen a paquets diferents del de la classe de la que s'hereta) usant el modificador **protected**. Si ho apliquem a la classe **Figura**, podem deixar només el seu constructor per defecte, ja que els atributs seran accessibles per la classe **Circulo**. La classe **Figura** queda com segueix:

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

En les Figures 6 i 7 apareix l'ús que es fa de l'herència estenent la classe **Figura**. Com es veu en les línies 7 i 8, la crida al constructor de la classe base (**super(x,y)**;) s'ha substituït per l'ús directe dels atributs heretats **x** i **y**, ja que ara són accessibles en la classe **Circulo**.

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

Figura 6: Classe Circulo

Figura 7: Classe Triangulo

Fixa't també que en la línia 7 s'usa **super.x** i en la línia 8 **this.y** sent ara equivalent l'ús de **super** i **this** sobre **x** i **y**, doncs aquests atributs també formen part d'un cercle. En les línies 23 i 24 desapareix tal diferència. En la línia 14 es manté la crida al mètode **toString()** de **Figura**, que encara

que com pots veure en la línia 30 no és necessari, sí que és recomanable la reutilització del mètode de la classe base. Per exemple: si es decideix canviar la paraula `Posicion` per `Punt` només s'ha de realitzar un canvi que es propagarà a les classes derivades sense alterar el codi d'aquestes.

La classe `GrupoFiguras`, redefinida en la Figura 8, ja no usa el tipus `Object` en l'`array`, sinó el tipus `Figura`. Això implica que els seus components només poden contindre objectes d'aquest tipus i els seus derivats `Circulo` i `Triangulo`. Encara que, si volem distingir-los haurem de seguir emprant la instrucció `instanceof`, ja no haurem de preocupar-nos de que les components de l'`array` continguin un objecte d'algun tipus que no siga una `Figura` o descendisca d'ella. Encara així, es podrien referenciar objectes de tipus `Figura` emprant el seu constructor per defecte. Com veurem en la següent secció, això es pot evitar fent de la classe `Figura` una classe abstracta. En aquesta classe també es defineix un únic mètode modificador i el mètode `toString()` es simplifica pel que fa a l'anterior disseny d'eixida sense exigir que les figures estiguen classificades. La nova definició del mètode `toString()` canvia l'execució de la classe `UsoDeGrupoFiguras` com es mostra a continuació:

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

```
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;}}

```

Figura 8: Classe `GrupoFiguras` emprant el tipus `Figura`

En la Figura 9 s'il·lustren les classes i relacions entre elles. Les classes es representen mitjançant caixes i les relacions entre elles mitjançant diferents tipus de línies i fletxes. Apareixen dos tipus de relacions.

D'una banda, la relació d'herència *ÉS-UN* representada amb una línia contínua i una fletxa sòlida des de la classe derivada fins a la classe base. Aquesta relació estableix una jerarquia de classes on la classe base és més



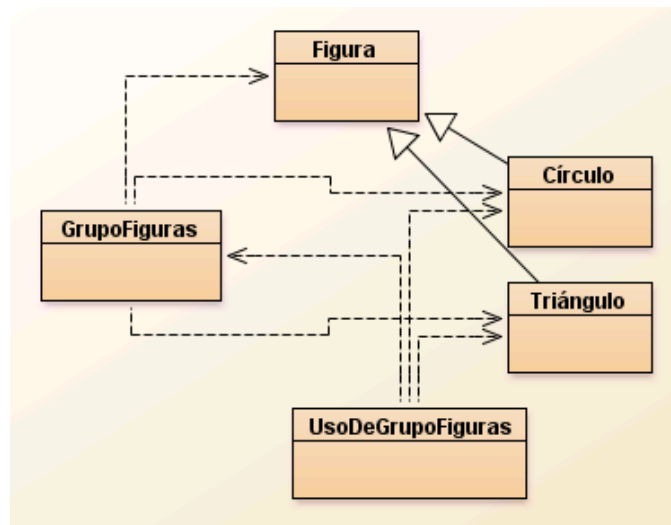


Figura 9: Relacions *ÉS-UN* i *USA-UN*

general que la derivada. D'altra banda, la relació *USA-UN*-tipus-de-dades representada per una fletxa amb línia discontinua. En el diagrama es representa que la classe **GrupoFiguras** usa el tipus **Figura**, **Circulo** i **Triangulo** però no deriva de cap. La classes **Circulo** i **Triangulo** no usen cap tipus de dades, sinó que ho hereten del tipus **Figura**, i la classe **UsoDeGrupoFiguras** usa les classes **Circulo**, **Triangulo** i **GrupoFiguras** però no usa la classe **Figura**.

**Exercici 3** *Defineix una classe Rectangulo amb atributs base i altura de tipus double que derive de Figura i amb els mateixos mètodes de Circulo i Triangulo. Canvia en alguna cosa GrupoFigura? Afegeix un rectangle al grup de figures definit en la classe UsoDeGrupoFiguras per a comprovar-ho.*

## 4 Classes abstractes i polimorfisme

En la secció anterior, es va controlar el tipus d'objectes que es podien incloure en l'array `listaFiguras`. Permetem només els objectes de la classe **Figura** i les seues derivades, però això implica que es pot crear una instància de **Figura** i emmagatzemar-se en l'array de figures

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

No obstant això, el propòsit inicial consistia a tindre un grup de cercles i triangles però no d'objectes de figures. Una forma d'evitar això consisteix a permetre que existisquen objectes de les classes **Circulo** i **Triangulo** però

no de `Figura`. Això s'aconsegueix definint aquesta última classe com a abstracta:

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

Fixa't com ha canviat el modificador en la definició dels atributs `x` i `y`. Amb aquest canvi permetem que aquest atribut siga visible només per a les seues classes derivades.

**Exercici 4** *Una altra forma d'evitar que s'afegeixen objectes de la classe `Figura` en l'array de figures consisteix en comprovar, mitjançant `instanceof`, el tipus de l'objecte que rep el mètode `anyadeFigura(Figura)` i inserir només els objectes el tipus dels quals descendeix de `Figura`. Què caldria fer cada vegada que s'escalara l'aplicació amb un nou tipus de figura? Ofeix l'herència algun tipus d'avantatge per al manteniment de l'aplicació?*

Les classes abstractes solen emprar-se per a desenvolupar una jerarquia de classes amb algun comportament comú.

**Exercici 5** *Es desitja que totes les figures disposen d'un mètode per a calcular la seua àrea. Defineix en la classe `Figura` un mètode abstracte `area()` que retorne un valor de tipus `double`.*

Després d'aquesta modificació de la classe `Figura` la implementació del mètode `area()` en les classes `Circulo` i `Triangulo` quedaria com apareix a continuació en la Figura 10.

<pre>public class Circulo     extends Figura{ protected double r;  public Circulo(double x,double y,     double radio){     this.x = x;     this.y = y;     r= radio;}  public double area (){     return Math.pow(r,2)Math.PI;}  public String toString (){     return "Circulo:"+         "\n\tPosición("+ x+", "+y+         ") \n\tRadio: "+r;} }}</pre>	<pre>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;     base= b;     altura = a;}  public double area (){     return base * altura / 2;}  public String toString){     return "Triángulo:"+         "\n\tPosición("+x+", "+y+         ") \n\tBase: "+base+         "\n\tAltura: "+altura;}}</pre>
---	---

Figura 10: Classes `Circulo` i `Triangulo`

En la Figura 11 s'il·lustra la jerarquia de les classes per a cercles i triangles. La classe `GrupoFiguras` usa el tipus `Figura` sense necessitat de canviar el seu codi i la classe de prova `UsoDeGrupoFiguras` també roman inalterada emprant les classes `Circulo`, `Triangulo` i `GrupoFiguras` per a crear objectes d'aquestes classes.

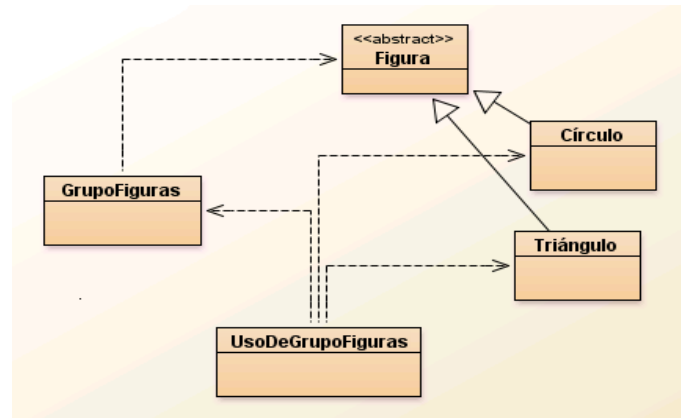


Figura 11: Relacions *ÉS-UN* i *USA-UN*

**Exercici 6** En definir el mètode `area()` en la classe `Figura`, les seues classes derivades han d'implementar aquest mètode. Implementa'l en totes les seues classes derivades incloent la classe `Rectangulo` que vas definir prèviament. Comprova l'àrea de les figures que es creen en la classe `UsoDeGrupoFiguras`.

**Exercici 7** Defineix un mètode `area()` en la classe `GrupoFiguras` que retorne la suma de les àrees de les figures d'un grup. Per a això recorre totes les figures referenciades en les components de l'atribut `listaFiguras` des de la posició zero fins a la posició `numF-1` aplicant el mètode `area()` a cada figura. Pots apreciar que l'herència ens proporciona polimorfisme de mètodes ja que per a cada tipus de figura s'executa el mètode que calcula la seua àrea.

Ara, podem seguir emprant l'herència per a estendre la jerarquia de classes que hem definit derivant una nova classe `Cilindro` a partir de la classe `Circulo` tal com s'il·lustra en la Figura 12.

En la Figura 13 apareix una implementació de la classe `Cilindro`. En aquesta classe s'afegeix un nou atribut `a` de tipus `double` que representa l'altura del cilindre. En aquesta classe es defineixen dos constructors. El primer (línies 3-5), rep quatre paràmetres: les dues coordenades de la posició, el radi de la base i l'altura. En la seua primera instrucció (línia 4) s'invoca a `super` per a no reescriure les assignacions que donen valor als atributs d'un cercle. El segon constructor (línies 6 i 7) rep dos paràmetres: un cercle `c`

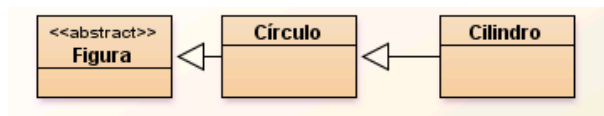


Figura 12: Diagrama de la jerarquia de classes

i una altura. En la línia 7 d'aquest constructor es crida al constructor de quatre paràmetres que s'ha definit en les línies 3-5. Per a això s'usa `this` com a nom del mètode anomenat. L'atribut `r` de `c` és accessible en aquesta classe ja que està definit com `protected` en la classe `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; } }
  
```

Figura 13: Classe Cilindro

**Exercici 8** *Implementa un mètode `area()` en la classe `Cilindro`. Aquest mètode ha de retornar l'àrea d'un cilindre sumant l'àrea d'un rectangle amb el doble de l'àrea d'un cercle. Crea un objecte de tipus `Rectangulo` la base del qual siga el perímetre del cercle calculat a partir del radi, i la seua altura la del cilindre. L'àrea del cercle es calcula invocant al mètode `area()` de `super`.*

## 5 Herència múltiple en Java

Suposem implementada la interfície `Volumen`:

```

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

i que la classe `Cilindro` es redefineix emprant aquesta interfície:

```

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 volum() { return super.area()*a; } }

```

En intentar compilar-la, rebem el següent missatge d'error

```

Cilindro is not abstract and does not override abstract method
superficie() in Volumen

```

Indicant-nos que tenim dues opcions: o declarem la nova classe com a abstracta o implementem el mètode `superficie()`. Com el mètode `volumen()` ja estava implementat, no produeix cap error.

**Exercici 9** *A què es deu aquest error? Fes els canvis pertinents per a solucionar-ho.*

Una vegada implementada tota la interfície, el diagrama de classes queda com en la Figura 14 en el qual pots apreciar que la implementació de l'interfície `Volumen` per la classe `Cilindro` es representa amb una línia discontinua amb una fletxa sòlida.

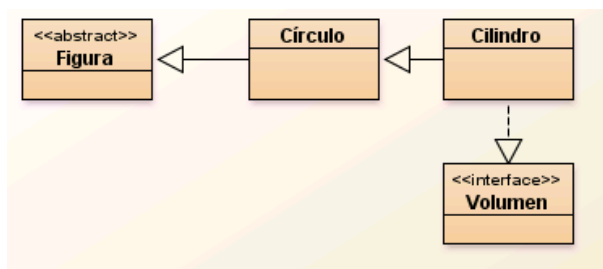


Figura 14: Diagrama de la jerarquia de classes i interfície

Definir variables d'un tipus interfície permet que aquestes variables referencien objectes de qualsevol classe que implemente la funcionalitat descrita en aquesta interfície. D'aquesta forma, podem estar segurs que aquests objectes disposen de les operacions que s'especifiquen en la interfície. En el següent exemple s'invoca al mètode `volumen()`:

```

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

```

Amb aquest ús de les interfícies es pot exigir una funcionalitat comuna a classes definides en diferents jerarquies de classes. Això implica un ús de l'herència independent de la jerarquia de classes.

**Exercici 10** *En la classe `GrupoFiguras` escriu un mètode `volumen()` que calcule la suma dels volums de totes les figures que implementen la interfície `Volumen`. Per a això, recorre totes les figures referenciades en les components de l'atribut `listaFiguras[]` acumulant el volum de les figures que implementen l'interfície `Volumen`. Per a comprovar si la classe d'un objecte implementa la interfície, pots emprar la instrucció `instanceof`*

## 6 Avaluació

L'assistència a les sessions de pràctiques és obligatòria per a aprovar l'assignatura. L'avaluació d'aquesta primera part de pràctiques es realitzarà mitjançant un examen individual en el laboratori.

## 7 Apèndix. Paquets i visibilitat

Quan els programes són relativament grans o es treballa en equip, és recomanable dividir el codi en parts o paquets per a estalviar temps de compilació davant modificacions, afavorir l'eficiència del treball en equip i controlar l'accés a les classes i interfícies evitant conflictes amb identificadors.

Els *paquets* són directoris que contenen fitxers amb classes precompilades (`.class`) seguint una jerarquia diferent a la seguida en l'herència. Normalment els paquets inclouen classes seguint un criteri de cohesió funcional. Per exemple, el paquet `java.awt.geom` conté classes per a definir i executar operacions sobre objectes relacionats amb la geometria en dues dimensions.

Les classes poden importar-se individualment o per paquets afegint el comodí `*` al final de la instrucció per a indicar que es desitja tindre accés a totes les classes del paquet. Les importacions s'expliciten al principi del fitxer. Per exemple

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

on els punts separen subpaquets, com es veurà més endavant. Aquesta notació també ens permet controlar l'espai de noms diferenciant les classes entre si quan tenen el mateix nom. Així, per exemple, en Java existeixen tres classes predefinides amb el mateix nom `Timer`, cadascuna en un paquet diferent i amb una funcionalitat totalment diferent. Les tres poden emprar-se en el mateix programa important-les al principi del fitxer:

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

i definint variables utilitzant la ruta per a arribar al directori on es troba la classe.

```
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();
```

També poden realitzar-se importacions estàtiques per a evitar els noms de les classes en la invocació de mètodes estàtics, referències a constants estàtiques, etc. Per exemple:

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

En un fitxer es pot definir més d'una classe però només una pot ser pública. En aquest cas, el nom del fitxer ha de coincidir amb el de la classe

pública (amb l'extensió `.java`). La resta de les classes del fitxer només seran visibles dins del paquet. És usual que els fitxers continguin una sola classe, però a voltes es declaren més classes quan només s'usen en les classes definides en aqueix mateix fitxer. Per a utilitzar paquets en Java, es necessita generar una estructura de directoris que tinga la mateixa jerarquia que les llibreries que es creen. Quan es vol que una classe pertanga a una llibreria, el nom del paquet al que pertany la classe ha d'especificar-se en la primera línia del fitxer amb la sintaxi

```
package paquet1.paquet2. ... .paqueteN;
```

La ruta de directoris descrita amb la notació punt, descriu el camí relatiu des del directori on es guarda tot el projecte fins al directori on s'emmagatzemà la classe. Per exemple, si tenim un projecte en el directori `lineales`, i es desitja que les classes definides en el fitxer `Pila.java` formen part d'un paquet `modelos`, el qual és un subpaquete del paquet `librerias`, s'escriuria la instrucció `package librerias.modelos;` en la primera línia del fitxer. Això equival a dir que les classes definides en el fitxer `Pila.java` estan disponibles en el directori `../lineales/librerias/modelos`.

Existeix un altre tema que concerneix als paquets i l'execució d'alguns dels comandos com el de compilació (`javac`), execució (`java`) i generació de documentació (`javadoc`). Per defecte, els comandos cerquen les llibreries a partir del directori on està instal·lat el JDK i del directori on s'executen. Quan es necessiten altres directoris, els comandos suposen l'existència de la variable d'entorn `CLASSPATH`, en la qual es defineix una seqüència de camins fins als directoris, a partir dels quals, els comandos cerquen els paquets.

L'accessibilitat a les classes no depèn només del paquet en el qual es troben, sinó també d'altres factors. Anem a revisar breument i de forma genèrica la visibilitat de les classes i dels seus components segons la definició que fem d'elles.

#### Definició de una classe

Sintaxi:

```
modifAcceso modifClase class NomClase  
                                [extends NomClase]  
                                [implements listaInterfaces]
```

on

**modifAcceso** indica des d'on es pot accedir a l'ús de la classe.

**public** la classe és visible des de qualsevol altra classe sense importar el paquet en el que estiga.

**sense especificar** accessible sòls a les classes del mateix paquet.

**private** sòls és visible en la classe en la que es defineix <sup>1</sup>.

---

<sup>1</sup>Es poden definir classes dins d'altres classes i se les coneix com *classes internes*.



**modifClase** afecta a les classes derivades de la classe.

**abstract** per a les classes abstractes.

**final** evita que la classe pugui ser derivada.

**Herència** tipus de classes de les quals es deriva.

**extends** s'utilitza per a indicar que la classe hereta de NomClase, en java només es permet heretar d'una única classe pare. En cas de no incloure la clàusula **extends**, s'assumirà que s'està heretant directament de la classe `java.lang.Object`

**implements** indica que aquesta classe és dels tipus d'interfície indicats per `listaInterfaces`, podent existir tants com vulguem separats per comes.

#### Definició de variables

Sintaxi:

[**modifVisibilidad**] [**modifAtributo**] tipo nomVariable;

on

**modifVisibilidad** delimiten l'accés des de l'exterior de la classe.

**public** accessible desde cualquier clase.

**private** sols és accessible des de la classe on es defineix.

**protected** accessible a les classes del mateix paquet en el qual es defineix així com en totes les seves classes derivades, fins i tot quan pertanyen a un altre paquet diferent.

*sense especificar* accessible des de qualsevol classe del mateix paquet.

**modifAtributos** característiques especials.

**static** la variable no forma part dels objectes sino que es comú a tots els objectes de la classe.

**final** el primer valor que rep la variable es inamovible.

**transient** exclueix l'atribut de la serialització <sup>2</sup> encara que la classe implemente la interfície **serializable**.

**volatile** informa que l'atribut és accessible de forma asíncrona per dos fils impedit que el compilador altere l'ordre de les instruccions.

#### Definició de mètodes de una classe

Sintaxi:

[**modifVisibilidad**] [**modifFunción**] tipo nomFunción (listaParámetros)

on

**modifVisibilitat** mateixes normes que els atributs.

**modifFunción** poden tindre els següents valors:

---

<sup>2</sup> Aplanar un objecte consisteix en obtenir la seua informació en forma de cadena de caràcters. El mètode `toString` de Java, es una ferramenta amb la que realitzar un aplanament definit per el programador. La serialització en Java es un aplanament amb una sintaxi predefinida. Sol utilitzar-se per transportar objectes a altres dispositius (disco,...) o per transmetre a través de la xarxa.

**static** es de la classe, no s'aplica als objectes. Quan s'invoca un mètode estàtic des de altra classe, es precedeix amb el nom de la classe en la que es defineix, seguit de un punt.

***sense especificar*** es un mètode dinàmic.

**final** el mètode no es pot sobreescriure en una classe derivada.

**abstract** es delega la implementació a una classe derivada.

**native** escrit en codi natiu<sup>3</sup> resultant d'alguna compilació.

**synchronized** executat en exclusivitat per un fil<sup>4</sup>. S'usa per al control de la concurrència.

---

<sup>3</sup>Codi natiu és aquell que és executable directament per un processador i es pot obtenir compilant un llenguatge d'alt nivell.

<sup>4</sup>Si diversos fils s'estan executant concurrentment i intenten executar al mateix temps un mètode dinàmic sobre el mateix objecte, si el mètode està marcat com `synchronized`, només un s'executa i els altres esperen al fet que acabe. Si el mètode és estàtic, només s'executa un mètode estàtic de la classe alhora. Un dels fils que espera prendrà el relleu en l'execució seguint una política d'assignació de l'exclusivitat.