
PRÁCTICAS DE LENGUAJES, TECNOLOGÍAS Y PARADIGMAS DE PROGRAMACIÓN.

PARTE I PROGRAMACIÓN EN JAVA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica I

Herencia y Polimorfismo en Java

Índice

1. Objetivo y planteamiento de la Práctica	2
2. Introducción	2
2.1. Repasando Java	2
2.2. Usando el tipo Object. Variables polimórficas	4
3. Herencia en Java	6
4. Clases abstractas y polimorfismo	9
5. Herencia múltiple en Java	12
6. Evaluación	14
7. Apéndice. Paquetes y visibilidad	15

1. Objetivo y planteamiento de la Práctica

El **objetivo** de esta primera parte de prácticas consiste en que utilices, madurándolos, los conocimientos vistos en teoría sobre herencia, polimorfismo y genericidad en el lenguaje Java. Esta primera parte la realizarás en tres sesiones divididas en dos prácticas: la primera práctica de dos sesiones está dedicada a aspectos básicos de herencia y polimorfismo en Java; la segunda práctica, de una sesión, está dedicada a los tipos genéricos. Las prácticas las realizarás usando el IDE BlueJ (<http://www.bluej.org/tutorial/tutorial-spanish-1.pdf>).

En esta primera práctica se te plantea la resolución de un problema mediante la que irás viendo diversos aspectos de la *herencia* y el *polimorfismo*. El **problema** es el siguiente: *diseñar una clase para guardar un grupo de figuras geométricas que contenga sólo círculos y triángulos, de forma que la incorporación de nuevos tipos de figuras suponga cambios mínimos en el software diseñado*. El problema tendrás que resolverlo de cuatro formas distintas. En la Sección 2 lo resolverás de dos formas: primero sin usar herencia ni polimorfismo y después sin herencia pero con variables polimórficas de tipo `Object`. En la Sección 3 lo resolverás usando herencia simple y variables polimórficas de tipo `Figura` definidas por el programador. En la Sección 4 lo tendrás que resolver usando clases abstractas y polimorfismo de métodos. Al final de esta sección, se amplía el problema a resolver añadiendo un nuevo tipo de figura *cilindro*. Esta ampliación se explica en la Sección 5 haciendo uso de herencia múltiple.

A lo largo del boletín de la práctica se proponen varios **ejercicios** que debes intentar realizar planteando tus dudas a tu profesor de prácticas. Asimismo, tu profesor de prácticas puede ampliar el problema a resolver en las sesiones de prácticas.

2. Introducción

2.1. Repasando Java

En esta práctica definirás clases reutilizando otras clases bajo el punto de vista de la reusabilidad del software. Se te proponen diversas opciones de implementación avanzando hacia soluciones más estables ante posibles modificaciones del problema. Además, se te plantean algunas posibilidades que ofrece el lenguaje para reutilizar las estructuras de datos y métodos haciendo uso de nuevos tipos de datos.

Se nos pide diseñar una clase para guardar información sobre dos tipos de figuras geométricas: círculos y triángulos. Para resolverlo podemos definir una clase para cada tipo de figura. La clase `Circulo` se define con dos atributos `x` e `y` para el punto en el que está situada la figura en un espacio de dos dimensiones, además contiene un atributo `r` que representa el radio

del círculo. Los tres atributos son del tipo real `double`. La clase `Triangulo` se define, al igual que los círculos, con dos coordenadas pero con los atributos `base` y `altura` también de tipo `double`. En la Figura 1 se ilustran estas definiciones y en ambas clases se define un constructor y el método `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: Definición de las clases `Circulo` y `Triangulo`

Ahora nos planteamos diseñar una clase `GrupoFiguras` donde guardar figuras de ambos tipos usando arrays tal y como aparece en 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: Definición de la clase `GrupoFiguras`

Como estas estructuras son homogéneas, definimos un array de círculos y otro de triángulos, cada uno con una capacidad de `MAX_NUM_FIGURAS/2`, como máximo. La cantidad de figuras de cada tipo queda representada mediante los atributos `numC` y `numT` respectivamente. Estos atributos se incrementan con cada inserción realizada por los métodos `anyadeCirculo(Circulo)` y `anyadeTriangulo(Triangulo)`. El método `toString()` recorre las figuras referenciadas en los arrays para mostrar la información de cada figura según

sus propios métodos `toString()`.

En la clase `UsoDeGrupoFiguras` de la Figura 3 se crea un grupo de figuras al que se le añade un círculo y un triángulo y se escribe la información del grupo.

```
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: Definición de la clase `UsoDeGrupoFiguras`

Su ejecución produce la salida:

```
----- Salida Estándar -----
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. Usando el tipo `Object`. Variables polimórficas

En el ejemplo de la sección anterior usamos dos arrays debido a la imposibilidad de almacenar objetos de distinto tipo en el mismo array. Este problema puede solventarse usando la clase `Object`, predefinida en Java, de la cual heredan todos los tipos definidos en el lenguaje y los definidos por el programador. Es decir, que todos los objetos son extensiones de este tipo. En la práctica, esto implica que las variables de este tipo pueden contener objetos de cualquier tipo, dando lugar a un polimorfismo de variables.

Veamos cómo usar este tipo para reducir la cantidad de arrays del ejemplo de la Figura 4. Este ejemplo es una redefinición de la clase `GrupoFiguras` en el que se utiliza un único array `listaFiguras` de tipo `Object` (línea 3). Esto permite almacenar en las componentes del array cualquier tipo de objeto, entre ellos, los de tipo `Circulo` y `Triangulo`, como puede verse en las asignaciones de `c` y `t` en los métodos `anyadirCirculo` y `anyadirTriangulo` (líneas 5 y 6). Estos dos métodos son los únicos públicos que alteran la estructura de datos y en sus parámetros formales sólo permiten círculos y triángulos que serán las únicas figuras guardadas. Si intentáramos añadir a un grupo de figuras `g` un objeto de tipo `Object` con una instrucción como

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

el compilador nos daría un error, ya que un tipo `Triangulo` es un `Object` pero no todo `Object` es un `Triangulo`. En este estado de la solución, ocurriría lo mismo si lo intentáramos con cualquier otro tipo distinto al de los parámetros.

Ejercicio 1 *Implementa las clases `Circulo`, `Triangulo`, `GrupoFiguras` y `UsoDeGrupoFiguras`. En esta última clase, añade varias figuras de ambos tipos, compila, ejecuta y observa el resultado. La clase `GrupoFiguras` has de definirla utilizando, en lo posible, el tipo `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: Clase `GrupoFiguras` usando el tipo polimórfico `Object`

Para mantener las figuras agrupadas por sus tipos en el valor de retorno del método `toString()`, se necesita distinguir entre los dos tipos de `Object` que se referencian en el array `listaFiguras`. En Java es posible preguntar si una instancia de una clase es de un tipo determinado mediante la instrucción `instanceof` usando la sintaxis:

```
variableObjeto instanceof NombreDeLaClase
```

Así, en las líneas 10 y 13 de la Figura 4 se pregunta por el tipo de los objetos para seleccionarlos.

El uso del tipo `Object` obliga al programador a preguntar constantemente por los tipos de las instancias y a realizar conversiones forzadas de tipo, dejando el uso correcto de los tipos en manos del programador. Esto último ocurrirá, por ejemplo, si en la clase `Circulo` sobrescribimos el método `equals` heredado de la clase `Object`:

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

donde el tipo de `c` se fuerza al tipo `Circulo` para que el compilador sepa que `c` hace referencia a un objeto que contiene un atributo `r`. Aquí se corre un alto riesgo ya que se da por supuesto que el parámetro formal `c` va a contener un círculo.

Ejercicio 2 *Sobrescribe el método `equals(Object)` para las clases `Circulo`, `Triangulo` y `GrupoFiguras`. Para ello supondremos que dos figuras son iguales si contienen exactamente los mismos valores en sus atributos, y que dos grupos de figuras son iguales si contienen las mismas figuras sin importar el orden ni la cantidad de veces que aparezcan. Prueba los métodos `equals(Object)` en `UsoDeGrupoFiguras` comparando objetos entre sí. ¿Qué ocurre si comparas figuras de tipos diferentes? Considera los cambios que realizarías en los métodos `anyadeCirculo`, `anyadeTriangulo` y `equals` de la clase `GrupoFiguras` si los grupos de figuras fueran conjuntos, es decir, sin elementos repetidos.*

3. Herencia en Java

Continuando con el ejemplo de las figuras, y observando que tanto los círculos como los triángulos están situados en una posición del plano, se puede sacar como factor común la posición de ambas y formar una nueva clase `Figura` como aparece en la Figura 5. La estructura de datos de los objetos de esta clase sólo contiene dos atributos `x` e `y` de tipo `double`, un método constructor y el método `toString()` para obtener su información.

```
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: Clase Figura

Ahora podemos extender la clase `Figura` con la Clase `Circulo`:

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

Como los atributos de la clase `Figura` son privados, no son visibles en otra clase incluidas sus derivadas. Si se quiere definir el constructor de un círculo con los dos parámetros de la posición `x` e `y`, y el parámetro del radio `r`, se deberá llamar al constructor de la clase base. Para referenciar a la clase base se usa la palabra reservada `super`. El constructor quedaría como:

```

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

```

donde en la línea 2 se llama al constructor de la clase base (**Figura**). Fíjate en que sólo se usa la palabra **super** y no el nombre de la clase base. Una restricción importante es que si se invoca al constructor de la clase base, ha de ser la primera instrucción del cuerpo del método. Además, si se sobrescribe algún método heredado, el de la clase base puede invocarse en la derivada con la notación punto, por ejemplo: **super.toString()**;

Se puede relajar la visibilidad de los atributos y métodos de una clase base para que sean visibles por todas sus clases derivadas (incluso cuando pertenezcan a paquetes distintos del de la clase de la que se hereda) usando el modificador **protected**. Si lo aplicamos a la clase **Figura**, podemos dejar sólo su constructor por defecto, ya que los atributos serán accesibles por la clase **Circulo**. La clase **Figura** queda como sigue:

```

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

```

En las Figuras 6 y 7 aparece el uso que se hace de la herencia extendiendo la clase **Figura**. Como se ve en las líneas 7 y 8, la llamada al constructor de la clase base (**super(x,y);**) se ha sustituido por el uso directo de los atributos heredados **x** e **y**, ya que ahora son accesibles en la clase **Circulo**.

<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>
--	--

Figura 6: Clase Circulo

Figura 7: Clase Triangulo

Fíjate también que en la línea 7 se usa `super.x` y en la línea 8 `this.y` siendo ahora equivalente el uso de `super` y `this` sobre `x` e `y`, pues estos atributos también forman parte de un círculo. En las líneas 23 y 24 desaparece tal diferencia. En la línea 14 se mantiene la llamada al método `toString()` de `Figura`, que aunque no es estrictamente necesaria (como puedes ver en la línea 30) sí que es recomendable ya que implica la reutilización del método de la clase base. Por ejemplo, si se decide cambiar la palabra `Posición` por `Punto` sólo se debe realizar un cambio que se propagará a las clases derivadas sin alterar el código de éstas.

La clase `GrupoFiguras`, redefinida en la Figura 8, ya no usa el tipo `Object` en el array, sino el tipo `Figura`. Esto implica que sus componentes sólo pueden contener objetos de este tipo y sus derivados: `Círculo` y `Triángulo`. Aunque, si queremos distinguirlos tendremos que seguir usando la instrucción `instanceof`, ya no tendremos que preocuparnos de que las componentes del array contengan un objeto de algún tipo que no sea una `Figura` o descienda de ella. Aún así, se podrían referenciar objetos de tipo `Figura` usando su constructor por defecto. Como veremos en la siguiente sección, esto puede evitarse haciendo de la clase `Figura` una clase abstracta. En esta clase también se define un único método modificador y el método `toString()`, que se simplifica con respecto al anterior diseño de salida sin exigir que las figuras estén clasificadas. La nueva definición del método `toString()` cambia la ejecución de la clase `UsoDeGrupoFiguras` como se muestra a continuación:

Salida Estándar
<pre>Círculo: Posición: (10.0, 5.0) Radio: 3.5 Triángulo: Posición(10.0,5.0) Base: 6.5 Altura: 32.0</pre>

```
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: Clase `GrupoFiguras` usando el tipo `Figura`

En la Figura 9 se ilustran las clases y relaciones entre ellas. Las clases

se representan mediante cajas y las relaciones entre ellas mediante diferentes tipos de líneas y flechas. Aparecen dos tipos de relaciones. Por un

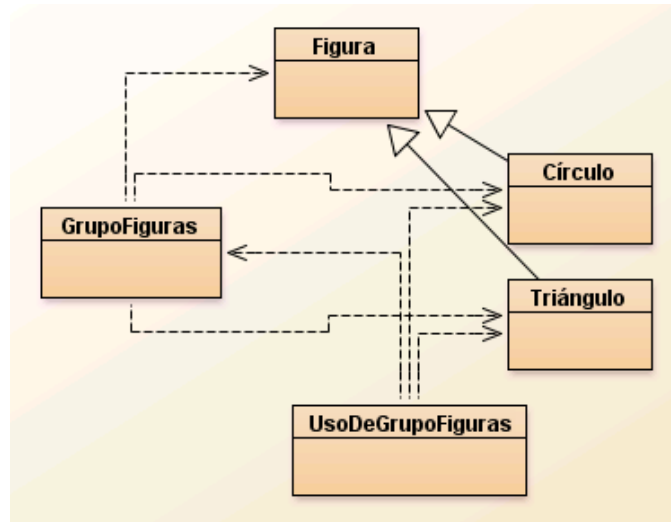


Figura 9: Relaciones *ES-UN* y *USA-UN*

lado, la relación de herencia *ES-UN* representada con una línea continua y una flecha sólida desde la clase derivada hasta la clase base. Esta relación establece una jerarquía de clases donde la clase base es más general que la derivada. Por otro lado, la relación *USA-UN*-tipo-de-datos representada por una flecha con línea discontinua. En el diagrama se representa que la clase *GrupoFiguras* usa el tipo *Figura*, *Circulo* y *Triangulo* pero no deriva de ninguna. Las clases *Circulo* y *Triangulo* no usan ningún tipo de datos, sino que lo heredan del tipo *Figura*, y la clase *UsoDeGrupoFiguras* usa las clases *Circulo*, *Triangulo* y *GrupoFiguras* pero no usa la clase *Figura*.

Ejercicio 3 Define una clase *Rectangulo* con atributos *base* y *altura* de tipo *double* que derive de *Figura* y con los mismos métodos de *Circulo* y *Triangulo*. ¿Cambia en algo *GrupoFiguras*? Añade un rectángulo al grupo de figuras definido en la clase *UsoDeGrupoFiguras* para comprobarlo.

4. Clases abstractas y polimorfismo

En la sección anterior, se acotó el tipo de objetos que se podían incluir en el array `listaFiguras`. Permitimos sólo los objetos de la clase *Figura* y sus derivadas, pero esto implica que se puede crear una instancia de *Figura* y guardarse en el array de figuras

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

Sin embargo, el propósito inicial consistía en tener un grupo de círculos y triángulos pero no de objetos figuras. Una forma de evitar esto consiste en permitir que existan objetos de las clases `Circulo` y `Triangulo` pero no de `Figura`. Esto se consigue definiendo esta última clase como abstracta:

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

```

Fíjate como ha cambiado el modificador en la definición de los atributos `x` e `y`. Con este cambio permitimos que este atributo sea visible para sus clases derivadas.

Ejercicio 4 *Otra forma de evitar que se añadan objetos de la clase `Figura` en el array de figuras consiste en comprobar mediante `instanceof` el tipo del objeto que recibe el método `anyadeFigura(Figura)` e insertar sólo los objetos cuyo tipo descende de `Figura`. ¿Qué habría que hacer cada vez que se escalara la aplicación con un nuevo tipo de figura? ¿La herencia ofrece algún tipo de ventaja para el mantenimiento de la aplicación?*

Las clases abstractas suelen usarse para desarrollar una jerarquía de clases con algún comportamiento común.

Ejercicio 5 *Se desea que todas las figuras dispongan de un método para calcular su área. Define en la clase `Figura` un método abstracto `area()` que devuelva un valor de tipo `double`.*

Tras esta modificación de la clase `Figura`, la implementación del método `area()` en las clases `Circulo` y `Triangulo` quedaría como aparece a continuación en la Figura 10.

En la Figura 11 se ilustra la jerarquía de las clases para círculos y triángulos. La clase `GrupoFiguras` usa el tipo `Figura` sin necesidad de cambiar su código y la clase de prueba `UsoDeGrupoFiguras` también permanece inalterada usando las clases `Circulo`, `Triangulo` y `GrupoFiguras` para crear objetos de dichas clases.

Ejercicio 6 *Al definir el método `area()` en la clase `Figura`, sus clases derivadas deben implementar este método. Impleméntalo en todas sus clases derivadas incluyendo la clase `Rectangulo` que definiste previamente. Comprueba el área de las figuras que se crean en la clase `UsoDeGrupoFiguras`.*

Ejercicio 7 *Define un método `area()` en la clase `GrupoFiguras` que devuelva la suma de las áreas de las figuras de un grupo. Para ello recorre todas las figuras referenciadas en las componentes del atributo `listaFiguras`*

```

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;}
}}

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;}}

```

Figura 10: Clases Circulo y Triangulo

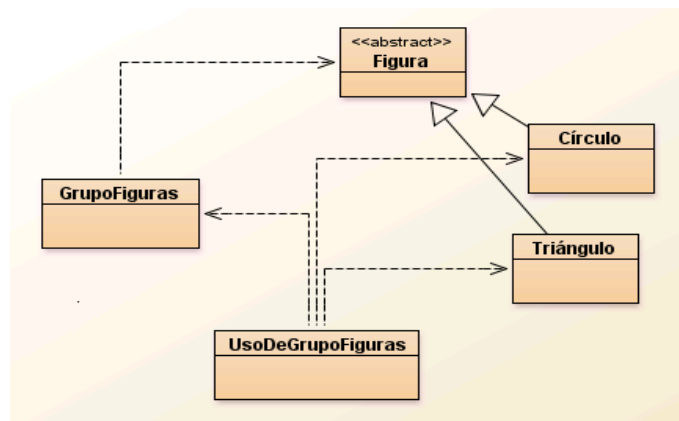


Figura 11: Relaciones *ES-UN* y *USA-UN*

desde la posición cero hasta la posición `numF-1` aplicando el método `area()` a cada figura. Puedes apreciar que la herencia nos proporciona polimorfismo de métodos ya que para cada tipo de figura se ejecuta el método que calcula su área.

Ahora, podemos seguir usando la herencia para extender la jerarquía de clases que hemos definido derivando una nueva clase `Cilindro` a partir de la clase `Circulo` tal y como se ilustra en la Figura 12.

En la Figura 13 aparece una implementación de la clase `Cilindro`. En esta

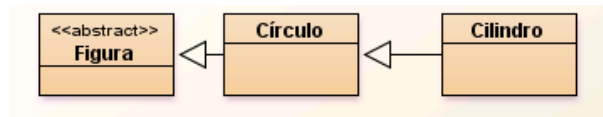


Figura 12: Diagrama de la jerarquía de clases

clase se añade un nuevo atributo **a** de tipo **double** que representa la altura del cilindro. En esta clase se definen dos constructores. El primero (líneas 3-5), recibe cuatro parámetros: las dos coordenadas de la posición, el radio de la base y la altura. En su primera instrucción (línea 4) se invoca a **super** para no reescribir las asignaciones que dan valor a los atributos de un círculo. El segundo constructor (líneas 6 y 7) recibe dos parámetros: un círculo **c** y una altura. En la línea 7 de este constructor se llama al constructor de cuatro parámetros que se ha definido en las líneas 3-5. Para ello se usa **this** como nombre del método llamado. El atributo **r** de **c** es accesible en esta clase ya que está definido como **protected** en la clase **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 volumen()
9         {return super.area()*a;}}
  
```

Figura 13: Clase Cilindro

Ejercicio 8 *Implementa un método **area()** en la clase **Cilindro**. Este método debe devolver el área de un cilindro sumando el área de un rectángulo con el doble del área de un círculo. Crea un objeto de tipo **Rectangulo** cuya base sea el perímetro del círculo calculado a partir del radio, y su altura la del cilindro. El área del círculo se calcula invocando al método **area()** de **super**.*

5. Herencia múltiple en Java

Supongamos implementada la interfaz **Volumen**:

```

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

y que la clase `Cilindro` se redefine usando esta interfaz:

```
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;}}
```

Al intentar compilarla, recibimos el siguiente mensaje de error

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

indicándonos que tenemos dos opciones: o declarar la nueva clase como abstracta o implementar el método `superficie()`. Además, como el método `volumen()` ya estaba implementado, no produce ningún error.

Ejercicio 9 *¿A qué se debe este error? Haz los cambios pertinentes para solucionarlo.*

Una vez implementa toda la interfaz, el diagrama de clases queda como en la Figura 14 en el que puedes apreciar que la implementación del interfaz `Volumen` por la clase `Cilindro` se representa con una línea discontinua con una flecha sólida.

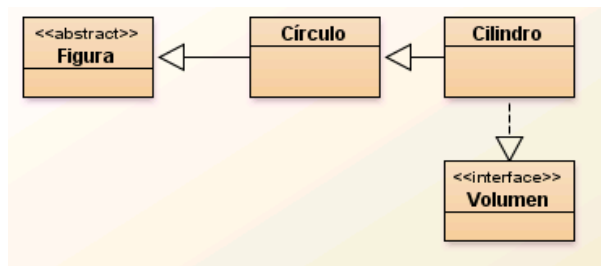


Figura 14: Diagrama de la jerarquía de clases e interfaz

Definir variables de un tipo interfaz permite que estas variables contengan objetos de cualquier clase que implemente la funcionalidad descrita en esa inferfaz. De esta forma, podemos estar seguros de que dichos objetos disponen de las operaciones que se especifican en la interfaz. En el siguiente ejemplo se invoca al método `volumen()`:

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

Con este uso de las interfaces se puede exigir una funcionalidad común a clases definidas en distintas jerarquías de clases. Esto implica un uso de la herencia independiente de la jerarquía de clases.

Ejercicio 10 *En la clase `GrupoFiguras` escribe un método `volumen()` que calcule la suma de los volúmenes de todas las figuras que implementen la interfaz `Volumen`. Para ello, recorre todas las figuras referenciadas en las componentes del atributo `listaFiguras[]` acumulando el volumen de las figuras que implementan el interfaz `Volumen`. Para averiguar si la clase de un objeto implementa la interfaz, puedes usar la instrucción `instanceof`*

6. Evaluación

La asistencia a las sesiones de prácticas es obligatoria para aprobar la asignatura. La evaluación de esta primera parte de prácticas se realizará mediante un examen individual en el laboratorio.

7. Apéndice. Paquetes y visibilidad

Cuando los programas son relativamente grandes o se trabaja en equipo, es recomendable dividir el código en partes o paquetes para ahorrar tiempo de compilación ante modificaciones, favorecer la eficiencia del trabajo en equipo y controlar el acceso a las clases e interfaces evitando conflictos con identificadores.

Los *paquetes* son directorios que contienen ficheros con clases precompiladas (`.class`) siguiendo una jerarquía diferente a la de la herencia. Normalmente los paquetes incluyen clases siguiendo un criterio de cohesión funcional. Por ejemplo, el paquete `java.awt.geom` contiene clases para definir operaciones sobre objetos relacionados con la geometría en dos dimensiones.

Las clases pueden importarse individualmente o por paquetes añadiendo el comodín `*` al final de la instrucción para indicar que se desea tener acceso a todas las clases del paquete. Las importaciones se explicitan al principio del fichero. Por ejemplo

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

donde los puntos separan subpaquetes, como se verá más adelante. Esta notación también nos permite controlar el espacio de nombres diferenciando las clases entre sí cuando tienen el mismo nombre. Así, por ejemplo, en Java existen tres clases predefinidas con el mismo nombre `Timer`, cada una en un paquete distinto y con una funcionalidad totalmente distinta. Las tres pueden usarse en el mismo programa importándolas al principio del fichero:

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

y definiendo variables utilizando la ruta para llegar al directorio donde se encuentra la clase.

```
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();
```

También pueden realizarse importaciones estáticas para evitar los nombres de las clases en la invocación de métodos estáticos, referencias a constantes estáticas, etc. Por ejemplo:

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

En un fichero se puede definir más de una clase pero sólo una puede ser pública. En este caso, el nombre del fichero debe coincidir con el de la clase

pública (con la extensión `.java`). El resto de las clases del fichero sólo serán visibles dentro del paquete. Es usual que los ficheros contengan una sola clase, pero a veces se declaran más clases cuando sólo se usan en las clases definidas en ese mismo fichero. Para utilizar paquetes en Java, se necesita generar una estructura de directorios que tenga la misma jerarquía que las librerías que se crean. Cuando se quiere que una clase pertenezca a una librería, el nombre del paquete al que pertenece la clase ha de especificarse en la primera línea del fichero con la sintaxis

```
package paquete1.paquete2. ... .paqueteN;
```

La ruta de directorios descrita con la notación punto, describe el camino relativo desde el directorio donde se guarda todo el proyecto hasta el directorio donde se guardará la clase.

Por ejemplo, si tenemos un proyecto en el directorio `lineales`, y se desea que las clases definidas en el fichero `Pila.java` formen parte de un paquete `modelos`, el cual es un subpaquete del paquete `librerias`, se escribiría la instrucción `package librerias.modelos;` en la primera línea del fichero. Esto equivale a decir que las clases definidas en el fichero `Pila.java` están disponibles en el directorio `../lineales/librerias/modelos`.

Existe otro tema que concierne a los paquetes y la ejecución de algunos de los comandos como el de compilación (`javac`), ejecución (`java`) y generación de documentación (`javadoc`). Por defecto, los comandos buscan las librerías a partir del directorio donde está instalado el JDK y del directorio donde se ejecutan. Cuando se necesitan otros directorios, los comandos suponen la existencia de la variable de entorno `CLASSPATH`, en la que se define una secuencia de caminos hasta los directorios, a partir de los cuales, los comandos buscan los paquetes.

La accesibilidad a las clases no depende solo del paquete en el que se encuentren, sino también de otros factores. Vamos a revisar brevemente y de forma genérica la visibilidad de las clases y de sus componentes según la definición que hagamos de ellas.

Definición de una clase

Sintaxis:

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

donde

modifAcceso indica desde dónde se puede acceder al uso de la clase.

public la clase es visible desde cualquier otra clase sin importar el paquete en el que esté.

sin especificar accesible sólo a las clases del mismo paquete.

private sólo es visible en la clase en la que se define¹.

¹Se pueden definir clases dentro de otras clases y se las conoce como *clases internas*.

modifClase afecta a las clases derivadas de la clase.

abstract para las clases abstractas.

final evita que la clase pueda ser derivada.

Herencia tipos de clases de las que se deriva.

extends se utiliza para indicar que la clase hereda de NomClase, en java sólo se permite heredar de una única clase padre. En caso de no incluir la cláusula extends, se asumirá que se está heredando directamente de la clase java.lang.Object

implements indica que esta clase es de los tipos de interfaz indicados por listaInterfaces, pudiendo existir tantos como queramos separados por comas.

Definición de variables

Sintaxis:

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

modifVisibilidad delimitan el acceso desde el exterior de la clase.

public accesible desde cualquier clase.

private solo es accesible desde la clase donde se define.

protected accesible en las clases del mismo paquete en el que se define así como en todas sus clases derivadas, incluso cuando pertenecen a otro paquete distinto.

sin especificar accesible desde cualquier clase del mismo paquete.

modifAtributos características especiales.

static la variable no forma parte de los objetos sino que es común a todos los objetos de la clase.

final el primer valor que recibe la variable es inalterable.

transient excluye el atributo de la serialización ² aunque la clase implemente la interfaz **Serializable**.

volatile informa de que el atributo es accesible de forma asíncrona por dos hilos impidiendo que el compilador altere el orden de las instrucciones.

Definición de métodos de una clase

Sintaxis:

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

modifVisibilidad mismas normas que los atributos.

modifFunción puede tener los siguientes valores:

²Aplanar un objeto consiste en obtener su información en forma de cadena de caracteres. El método **toString** de Java, es una herramienta con la que realizar un aplanamiento definido por el programador. La serialización en Java es un aplanamiento con una sintaxis predefinida. Suele utilizarse para transportar objetos a otros dispositivos (disco,...) o para transmitirlo a través de la red.

static es de la clase, no se aplica a los objetos. Cuando se invoca un método estático desde otra clase, se precede con el nombre de la clase en la que se define, seguido de un punto.

sin especificar es un método dinámico.

final el método no se puede sobrescribir en una clase derivada.

abstract se delega la implementación a una clase derivada.

native escrito en código nativo³ resultante de alguna compilación.

synchronized ejecutado en exclusividad por un hilo ⁴. Se usa para el control de la concurrencia.

³Código nativo es aquel que es ejecutable directamente por un procesador y se puede obtener compilando un lenguaje de alto nivel.

⁴Si varios hilos se están ejecutando concurrentemente e intentan ejecutar al mismo tiempo un método dinámico sobre el mismo objeto, si el método está marcado como synchronized, sólo uno se ejecuta y los demás esperan a que acabe. Si el método es estático, sólo se ejecuta un método estático de la clase a la vez. Uno de los hilos que espera tomará el relevo en la ejecución siguiendo una política de asignación de la exclusividad.