Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2020-21

Parte I: Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica I

Polimorfismo en Java: herencia y sobrecarga

Índice

1. Introducción

En esta práctica definirás clases reutilizando otras clases bajo el punto de vista de la reusabilidad del software. Se proponen diversas opciones de implementación avanzando hacia soluciones más estables ante posibles modificaciones del problema.

1.1. El problema

Se nos pide diseñar una clase para guardar información de dos tipos de figuras geométricas: círculos y triángulos. Previamente necesitaríamos definir una clase para cada tipo de figura.

La clase `Circle` se define con dos atributos `x` e `y`, para indicar el punto donde esté la figura en un espacio bidimensional, y un atributo `radius` que representa su radio. La clase `Triangle` se define, al igual que `Circle`, con dos coordenadas `x` e `y`, pero con los atributos `base` y `height`, correspondientes a su tamaño. En la Figura 1 se ilustran estas definiciones, y en ambas clases se define un constructor y los métodos `equals(Object)` y `toString()`.

```
public class Circle {
 private double x, y;
 private double radius;

 public Circle(double a,
        double b, double c){
   x = a; y = b; radius = c;
 }

 public boolean equals(Object o){
   if (!(o instanceof Circle)) {
      return false; }
   Circle c = (Circle) o;
   return x == c.x && y == c.y &&
   radius == c.radius;
 }

 public String toString(){
   return ``Circle:\n\t'' +
     ``Position: ('' +x+ ``,'' +
     y+``)\n\tRadius: '' +radius;
 }
}
```

```
public class Triangle {
  private double x, y;
  private double base, height;

  public Triangle(double a,
     double b, double c, double d){
    x = a; y = b;
    base = c; height = d;
  }
  public boolean equals(Object o){
    if (!(o instanceof Triangle)){
       return false; }
    Triangle t = (Triangle) o;
    return x == t.x && y == t.y &&
           base == t.base &&
           height == t.height;
  }
  public String toString(){
    return ``Triangle:\n\t'' +
      ``Position: ('' +x+ ``,'' +
      y+``)\n\tBase: '' +base+
      ``\n\tHeight: '' +height;
  }
}
```

Figura 1: Definición de las clases `Circle` y `Triangle`, sin herencia

NOTA: Sobrecarga de métodos. La implementación de los métodos `equals(Object)` y `toString()` es una sobreescritura de los mismos, heredados de la clase `Object`.[1]

Como la cabecera del método `equals` no restringe el tipo del parámetro de entrada, es recomendable usar `instanceof` para tratar el caso en el que el objeto recibido no sea una instancia de la misma clase. Se recuerda que en Java se puede preguntar si una instancia de una clase es de un tipo determinado mediante la instrucción `instanceof` usando la sintaxis:

```
variableObjeto instanceof NombreDeLaClase
```

Así, la primera instrucción en las implementaciones del método `equals` de la Figura 1 permite devolver el valor `false` si el objeto recibido no es un círculo o triángulo, respectivamente. La segunda instrucción (**coerción explícita** o casting a `Circle` y a `Triangle`, respectivamente) es necesaria para disponer de una referencia de dichas clases y, por tanto, tener visibilidad de sus atributos (lo que no sería posible con la referencia de tipo `Object`). El casting resulta ser una operación segura al realizarse solamente después de la verificación de la instancia mediante `instanceof`.

Importancia del casting.
Reflexiona sobre los problemas potenciales que plantearía la siguiente implementación del método para la clase `Circle`:

```
public boolean equals(Object o) {
  return x == ((Circle)o).x && y == ((Circle)o).y &&
         radius == ((Circle)o).radius;
}
```

Ya tenemos la definición de las dos clases que representan dos tipos de figuras. Ahora deberíamos plantear una solución para almacenar un grupo de figuras que pueden ser triángulos o círculos.

Podríamos plantear una solución usando un array de elementos `Object` haciendo uso de la coerción y comprobación de tipos para garantizar el correcto funcionamiento. Sin embargo, esta sería una solución poco recomendable desde el punto de vista de la ingeniería del software[2] (mantenibilidad, extensibilidad, etc.).

A continuación veremos dos soluciones que explotan las características del lenguaje (herencia y variables polimórficas).

[1] En Java existe una jerarquía de clases ya definida y la raíz de dicha jerarquía es la clase `Object`, por lo que toda clase hereda de ésta.

[2] Estos aspectos se verán con mayor profundidad en la asignatura de tercer curso, "Ingeniería del software".

2. Solución 1: Usando herencia y sobrecarga

La herencia nos permite definir una jerarquía de clases agrupando las características y comportamiento comunes a varias clases en una clase *padre*. En nuestro problema, observamos que tanto los círculos como los triángulos están situados en una posición del plano, por lo que tiene sentido crear una nueva clase `Figure` que caracterice la posición y de la que heredarán los dos tipos de figura. Podemos ver su definición en la Figura 2. Contiene dos atributos `x` e `y` de tipo `double`, un método constructor y los métodos `equals(Object)` y `toString()`.

```
public class Figure {
  private double x, y;
  public Figure(double x, double y) {
    this.x = x; this.y = y;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Figure)) { return false; }
    Figure f = (Figure) o;
    return x == f.x && y == f.y;
  }
  public String toString() {
    return ``Position: (`` + x + ``, `` + y + ``)``;
  }
}
```

Figura 2: Clase `Figure`

Ahora podemos extender la clase `Figure` con la clase `Circle`. En otras palabras, `Circle` heredará de `Figure`:

```
public class Circle extends Figure {
  private double radius;
  ...
}
```

Como los atributos de la clase `Figure` son privados, no son visibles en otras clases, incluidas sus derivadas. Una posibilidad para definir el constructor de `Circle` (dado que no tenemos acceso a los atributos `x` e `y`), es usando el constructor de la clase base (de la clase `Figure`) mediante la palabra reservada `super`. El constructor sería:[3]

```
public Circle(double x, double y, double r) {
  super(x, y);
  radius = r;
}
```

[3] El código completo puede verse en la Figura 3.

Si se invoca al constructor de la clase base, dicha llamada debe ser la primera instrucción del cuerpo del constructor de la subclase.

> **NOTA:** La palabra clave `super` también se usa para hacer referencia a métodos de la clase padre que han sido sobrescritos en la subclase, por ejemplo: `super.toString();`

En la Figura 3 podemos ver cómo queda el código completo de las clases `Circle` y `Triangle` definidas como subclases de `Figure`.

```
public class Circle
   extends Figure {
 private double radius;

 public Circle(double x,
  double y, double r) {
    super(x, y);
    radius = r;
 }

 public String toString() {
  return ``Circle:\n\t'' +
         super.toString() +
         ``\n\tRadius: '' +
         radius;
 }
}
```

```
public class Triangle
   extends Figure {
 private double base, height;

 public Triangle(double x,
  double y, double b, double h) {
    super(x, y);
    base = b;
    height = h;
 }
 public String toString() {
  return ``Triangle:\n\t'' +
         super.toString() +
         ``\n\tBase: '' + base +
         ``\n\tHeight: '' + height;
 }
}
```

Figura 3: Nueva definición de las clases `Circle` y `Triangle`

Visibilidad de atributos y reutilización

Se podría relajar la visibilidad de los atributos y métodos de una clase base para que fueran visibles en todas sus clases derivadas (incluso cuando pertenezcan a paquetes distintos del de la clase de la que se hereda) usando el modificador `protected`, en lugar de `private`.

Un uso adecuado de la herencia consiste en reutilizar al máximo todo lo declarado en la superclase. Por eso en la solución propuesta se invoca al constructor de la clase `Figure` en los constructores de las subclases (lo cual es compatible con la declaración de los atributos como `protected` o como `private`) en lugar de asignar valores a los atributos heredados (esta solución sería posible solo si los atributos son `protected`).

Este mismo principio se aplica cuando sobrescribimos métodos: si lo implementado en la superclase es aplicable en las subclases, es preferible invocar al método de la superclase. Podemos ver un ejemplo en el código para el método `toString()`, en las implementaciones de las clases `Circle` y `Triangle` (Figura 3).

Ejercicio 1 *Crea un proyecto BlueJ de nombre* `ltp`*. En el mismo, crea un paquete de nombre* `practica1`*. Añade a este paquete las clases que te proporcionamos en Poliformat como punto de partida del trabajo de esta práctica.*

Ejercicio 2 *El método* `equals` *definido en la clase* `Figure` *establece que dos figuras son iguales cuando tienen la misma posición. Debemos refinar este método para las subclases* `Circle` *y* `Triangle`*:*

Sobrescribe el método `equals(Object)` *para las clases* `Circle` *y* `Triangle` *de manera que reutilices lo ya implementado en la clase* `Figure`*.*

Definiendo el grupo de figuras

Una vez definida la jerarquía de clases para figuras, el problema a resolver es el de definir un grupo de figuras que pueda contener tanto triángulos como círculos. La clase `FiguresGroup` (definida en la Figura 4) usa un array de tipo `Figure`. Esto implica que sus componentes solo pueden contener objetos de este tipo (clase) y de sus tipos (clases) derivados: `Circle` y `Triangle`.

```
public class FiguresGroup {
  private static final int NUM_FIGURES = 10;  // constante
  private Figure[] figuresList = new Figure[NUM_FIGURES];
  private int numF = 0;
  public void add(Figure f) { figuresList[numF++] = f; }
  public String toString() {
    String s = ``'';
    for(int i = 0; i < numF; i++) s += ``\n'' + figuresList[i];
    return s;
  }
}
```

Figura 4: Clase `FiguresGroup` usando el tipo `Figure`

Si necesitáramos distinguir de qué tipo concreto es un objeto del array, tendremos que usar la instrucción `instanceof`, pero no nos preocuparemos de que las componentes del array contengan un objeto de algún tipo que no sea `Figure` o descienda de ella. Aún así, se podrían referenciar objetos de tipo `Figure` (como veremos en la segunda solución, esto puede evitarse haciendo abstracta la clase `Figure`).

Se introduce la siguiente clase `FiguresGroupUse` (definida en la Figura 5) como ejemplo de uso de la clase `FiguresGroup`. La ejecución de esta clase se muestra en la figura siguiente.

```java
public class FiguresGroupUse {
  public static void main(String[] args) {
    FiguresGroup g = new FiguresGroup();
    g.add(new Circle(10, 5, 3.5));
    g.add(new Triangle(10, 5, 6.5, 32));
    System.out.println(g);
  }
}
```

Figura 5: Definición de la clase `FiguresGroupUse`

```
Circle:
    Position: (10.0, 5.0)
    Radius: 3.5
Triangle:
    Position: (10.0, 5.0)
    Base: 6.5
    Height: 32.0
```

Figura 6: Salida Estándar de `FiguresGroupUse`

Igualdad de grupos de figuras

Consideramos que dos grupos de figuras son iguales si contienen las mismas figuras, sin importar el orden ni la cantidad de veces que aparezcan en el grupo. Es decir, bastará con comprobar si cada figura contenida en un grupo se encuentra también en el otro, y viceversa.

En la anterior implementación de la clase `FiguresGroup` no se define un método `equals(Object)`. En el siguiente ejercicio deberás implementar dicho método para la nueva clase (sobrecarga). Para simplificar el trabajo, daremos la implementación de dos métodos (privados de la clase) que podrás utilizar para implementar `equals` (ver la figura 7).

El método `found(Figure)` verifica si la figura recibida como argumento se encuentra, o no, en el grupo de figuras que invoca al método. Aprecia el uso del `equals` en la expresión `figuresList[i].equals(f)`: si hay una referencia a un objeto `Circle` en la posición `i` del array, entonces se invoca el método `equals` de la clase `Circle`; pero si hay un objeto `Triangle`, entonces se ejecutará el código del método `equals` de la clase `Triangle`. Y esta decisión se tomará en tiempo de ejecución: se trata de un caso de polimorfismo con enlace dinámico.

El método `included(FiguresGroup)` verifica si el grupo `g` recibido como argumento está incluido, o no, en el grupo de figuras que invoca al método, `this`. Para ello, comprueba si cada una de las figuras que contiene `g` se encuentra en `this`.

Ejercicio 3 *Sobrescribe el método* `equals(Object)` *para la clase* `FiguresGroup`*,*

```
public class FiguresGroup {
  ...
  private boolean found(Figure f) {
    for(int i = 0; i < numF; i++) {
       if (figuresList[i].equals(f)) return true;
    }
    return false;
  }
  private boolean included(FiguresGroup g) {
    for(int i = 0; i < g.numF; i++) {
       if (!found(g.figuresList[i])) return false;
    }
    return true;
  }
}
```

Figura 7: Métodos auxiliares de `FiguresGroup`

teniendo en cuenta que dispones de los métodos `included(FiguresGroup)` *y* `found(Figure)` *en la misma clase.*

Prueba el método implementado invocándolo en `FiguresGroupUse` *y comparando objetos entre sí.*

Ejercicio 4 *Considera los cambios que realizarías en los métodos* `add` *y* `equals` *de la clase* `FiguresGroup` *si los grupos de figuras fueran conjuntos, es decir, sin elementos repetidos. No es necesario que lo implementes.*

En la Figura 8 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 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 `FiguresGroup` usa el tipo `Figure`, pero no deriva de ninguna. La clases `Circle` y `Triangle` no usan ningún tipo de datos, sino que lo heredan del tipo `Figure`, y la clase `FiguresGroupUse` usa las clases `Circle`, `Triangle` y `FiguresGroup` pero no usa la clase `Figure`.

Ejercicio 5 *Para representar la figura geométrica Rectángulo, define una clase* `Rectangle` *con atributos* `base` *y* `height` *de tipo* `double`*, que derive de* `Figure`*, y con los mismos métodos que* `Circle` *y* `Triangle`*. ¿Cambia en algo* `FiguresGroup`*? Añade un rectángulo al grupo de figuras definido en la clase* `FiguresGroupUse` *para comprobarlo.*

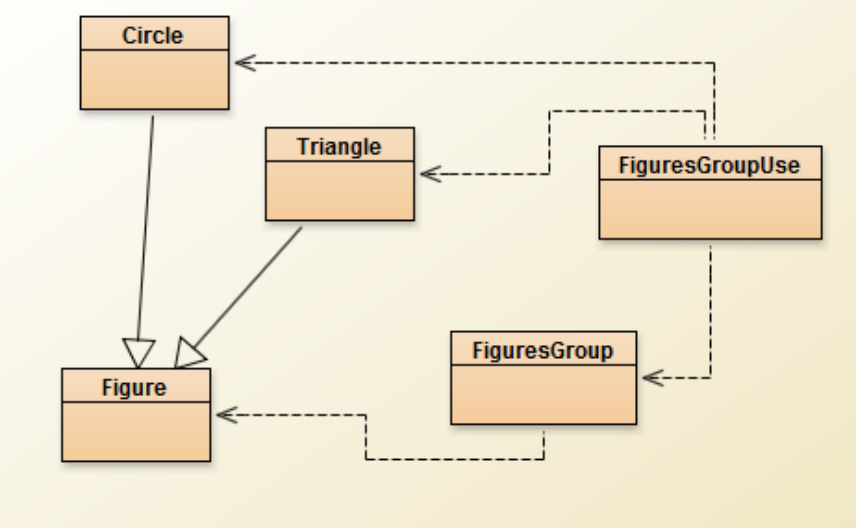

Figura 8: Relaciones *ES-UN* y *USA-UN*

Ejercicio 6 *Para representar la figura geométrica Cuadrado, define una clase* `Square`*, sin atributos, que derive de* `Rectangle`*, y con la misma funcionalidad que su superclase. ¿Has necesitado sobrescribir algún método de* `Rectangle`*? ¿Por qué?*

3. Solución 2: Usando clases abstractas

En la sección anterior se acotó el tipo de objetos que se podían incluir en el array `figuresList`. Permitimos solo los objetos de la clase `Figure` y sus derivadas, pero esto implica que se puede crear una instancia de `Figure` y guardarla en el array de figuras mediante, por ejemplo:

```
figuresList[pos] = new Figure(10, 5);
```

Sin embargo, nuestro propósito inicial era tener un grupo de figuras geométricas concretas (círculos, triángulos, rectángulos, cuadrados, y cualesquiera otras que se declaren) pero no de objetos `Figure`. Una forma de evitar esto consiste en permitir que existan objetos de las clases derivadas pero no de `Figure`. Esto se consigue definiendo esta última clase como abstracta. Las clases abstractas suelen usarse para desarrollar una jerarquía de clases con algún comportamiento común.

```
public abstract class Figure {
  private double x, y;
  ...
}
```

Se podría cambiar los modificadores en la definición de atributos[4].

Ejercicio 7 *Observa de nuevo la clase* `FiguresGroup` *de la Figura 4. Podríamos evitar que se añadan objetos de la clase* `Figure` *en el array de figuras comprobando mediante* `instanceof` *el tipo del objeto que recibe el método* `add(Figure)` *e insertar solo los objetos cuyo tipo desciende de* `Figure`.

Es una solución claramente peor, y no se pide implementarla. ¿Cómo se vería afectado este método si se definieran nuevos tipos de figuras (derivadas de `Figure`*)? ¿La herencia ofrece algún tipo de ventaja para el mantenimiento de la aplicación en este caso?*

Ejercicio 8 *Se desea que todas las figuras dispongan de un método para calcular su área. Define en la clase* `Figure` *un método abstracto* `area()` *que devuelva un valor de tipo* `double`.

Tras esta modificación de la clase `Figure`, las clases derivadas dejarán de compilar, con el mensaje de error: *"... is not abstract and does not override abstract method area() in Figure"*.

Ejercicio 9 *Resuelve el problema implementando el método* `area()` *en cada una de las subclases de* `Figure`.

Recordamos que el área del círculo podemos calcularla como la expresión `Math.PI * radius * radius`*, la del triángulo como* `base * height / 2`*, y la del rectángulo mediante* `base * height`.

Modifica el código de `FiguresGroupUse` *para calcular el área de las figuras que se crean en la misma.*

En la Figura 9 se ilustra la jerarquía de las clases para las figuras geométricas consideradas. La clase `FiguresGroup` usa el tipo `Figure` sin necesidad de cambiar su código, y la clase de prueba `FiguresGroupUse` también permanece inalterada usando `FiguresGroup` y las clases derivadas de `Figure` para crear objetos de dichas clases.

Ejercicio 10 *Define un método* `area()` *en la clase* `FiguresGroup` *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* `figuresList` *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.*

[4]En la definición de los atributos `x` e `y` se podría haber especificado `protected` en lugar de `private`. Si se usa el modificador `protected` estos atributos serán visibles para sus clases derivadas. Si se usa el modificador `private` no serán visibles para sus clases derivadas. En ambos casos los atributos existirán en los objetos instanciados. La elección del modificador dependerá de cómo se diseñen las clases, y es algo independiente de que la superclase sea abstracta o no.

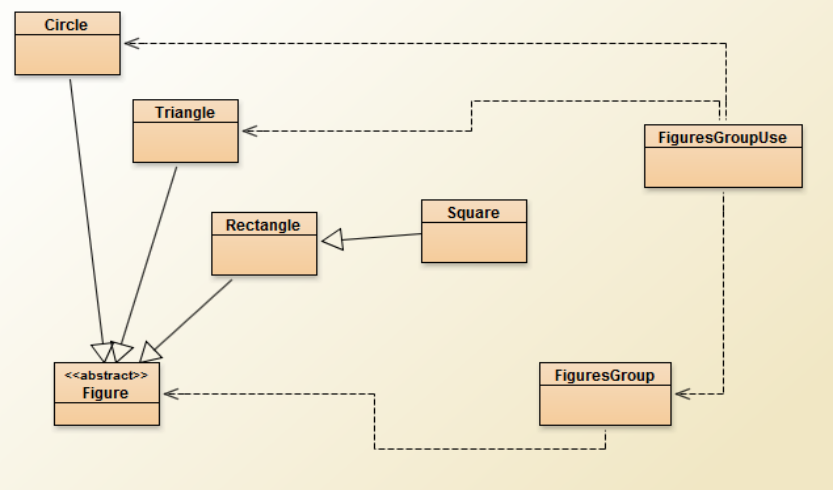

Figura 9: Relaciones *ES-UN* y *USA-UN*

Ejercicio 11 *Define un método* `greatestFigure()` *en la clase* `FiguresGroup` *que devuelva la figura del grupo cuya área sea mayor. De nuevo, tendrás que recorrer todas las figuras en el array* `figuresList` *y aplicar el método* `area()` *a cada figura.*

Ampliación

Se os proponen los siguientes ejercicios como ampliación de esta práctica. Resolverlos se considera ampliación por cuanto podrían exceder el tiempo de la sesión de laboratorio.

Ejercicio 12 *Se desea que todas las figuras dispongan de un método para calcular su perímetro. Define en la clase* `Figure` *un método abstracto* `perimeter()` *que devuelva un valor de tipo* `double`*.*

Ejercicio 13 *El método* `perimeter()` *debe ser implementado en cada subclase de* `Figure`*. Recuerda que el perimetro del círculo se puede calcular mediante* `2 * Math.PI * radius`*, el del triángulo mediante la suma de las longitudes de sus tres lados, y el del rectángulo mediante* `2 * (base + height)`*. Puedes apreciar que no dispones de información suficiente para calcular el perimetro del triángulo. ¿Qué solución piensas que podrías dar a este problema? ¿Declarar* `Triangle` *como clase abstracta? ¿No calcular el perímetro real, sino devolver un valor especial, -1, por ejemplo? ¿Modificar la declaración de atributos de la clase? Justifica tu elección.*