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

Parte I
Programación en Java

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Práctica 3

Polimorfismo e interfaces en Java

Índice

1. Objetivo / planteamiento

En esta tercera práctica se plantea la ampliación de la solución que se realizó en la primera práctica. La ampliación consiste en el uso de un tipo de clases en *Java*: las *interfaces*[1]. En concreto, en los ejercicios de esta práctica, deberás añadir funcionalidades adicionales a las clases que implementaste entonces, en tres pasos: usando una interfaz predefinida, extendiendo una interfaz, y diseñando una interfaz.

2. Uso de una interfaz predefinida

2.1. La interfaz `Comparable`

Como se comentó en el material de lectura previa de la práctica 2, existe una interfaz de uso común para comparar objetos de una clase entre sí: `Comparable`. Esta interfaz especifica el método `int compareTo(T)`, cuyo parámetro es de tipo genérico. La norma establece que el resultado de aplicar este método a un objeto `o1` recibiendo como parámetro otro objeto `o2` (`o1.compareTo(o2)`) debe ser:

- si `o1` = `o2` el resultado de la comparación es el valor cero.
- si `o1` < `o2` el resultado de la comparación es un valor entero negativo.
- si `o1` > `o2` el resultado de la comparación es un valor entero positivo.

Esta norma puede concretarse para cada clase que la implemente. La comparación entre figuras no tiene un orden natural, pero se puede definir un orden entre figuras usando su área (tamaño). Se decide instanciar la norma anterior de la siguiente manera: dadas dos figuras `f1` y `f2` de un tipo derivado de `Figure` la comparación

```
f1.compareTo(f2)
```

devuelve los siguientes valores:

- si `f1.area()` = `f2.area()` el resultado es el valor cero.
- si `f1.area()` < `f2.area()` el resultado es un valor entero negativo.
- si `f1.area()` > `f2.area()` el resultado es un valor entero positivo.

Ejercicio 1 *En el proyecto* `ltp`*, crea un paquete de nombre* `practica3`*. Añade a este paquete una copia de todas las clases implementadas en la primera práctica (disponibles en la carpeta* `ltp/practica1`*). Así, podrás resolver los ejercicios siguientes sobre las clases del paquete* `practica3`*, sin modificar ni perder nada de lo desarrollado en el paquete* `practica1`*.*

[1] Se recomienda consultar el anexo o material de lectura previa de la práctica 2 acerca de las *interfaces*.

Ejercicio 2 *Realiza los cambios necesarios en la clase* `Figure` *de forma que pueda determinarse cuando una figura es mayor (más grande) que otra. Esta clase debe implementar el interfaz* `Comparable` *para que todos los objetos de sus clases derivadas sean comparables entre sí. En esta implementación, el tipo genérico* `T` *de la interfaz debe concretarse en el tipo de una clase (en este caso,* `Figure`*).*

Una alternativa de diseño, menos acertada a la propuesta en el ejercicio anterior, hubiera consistido en implementar la interfaz solo en clases concretas (en `Circle`, `Triangle`, etc.). Es decir, que cada figura hubiera implementado su propio método `compareTo`. En ese caso, solo se hubieran podido comparar pares de objetos de la misma subclase, y no figuras de cualquier tipo concreto. Además, esto hubiera dado lugar a métodos idénticos al que has implementado en el ejercicio anterior repartidos por todas las subclases. La solución al ejercicio que has realizado es la deseable en aras de un buen diseño (más sostenible desde el punto de vista del mantenimiento de la aplicación).

2.2. La interfaz `List`

En el citado material de lectura previa de la práctica 2, también se comentó que uno de los usos de las interfaces consiste en la especificación de TAD's. En el paquete `java.util` de *Java* está definida la interfaz `List` que especifica las operaciones que debe implementar una clase para poder ver sus elementos como una lista. Las listas tienen un tamaño variable y sus elementos ocupan posiciones numeradas consecutivamente con números enteros a partir de la posición 0. Algunas de las operaciones de esta interfaz son:

- `void add(int index, Object element)` que inserta el elemento del segundo parámetro en la posición indicada en el primer parámetro. `IndexOutOfBoundsException` es una de las excepciones que lanza este método, en concreto, se lanza para el caso en que la posición no exista.
- `void add(Object element)` funciona como el anterior pero el elemento se añade al final de la lista.
- `Object get(int index)` aplicado a una lista devuelve el elemento que ocupa la posición indicada por su parámetro. Si no existe tal posición, lanza la excepción `IndexOutOfBoundsException`.
- `int size()` devuelve la cantidad de elementos de la lista.

Las clases `LinkedList` y `ArrayList`, definidas en el paquete `java.util`, implementan la interfaz `List`. Cada una implementa las operaciones anteriores teniendo en cuenta sus propias estructuras de datos:

- `ArrayList`: Su implementación se basa en un array redimensionable que duplica su tamaño cada vez que se necesita más espacio.
- `LinkedList`: Esta implementación se basa en una lista doblemente enlazada donde cada nodo tiene una referencia al anterior y al siguiente nodo.

Si se quisiera que los objetos de la clase `FiguresGroup` pudieran manejarse como una lista, esta clase podría implementar la interfaz `List`. Esto implicaría escribir el código de todos los métodos como hacen las clases `ArrayList` y `LinkedList`.

Otra alternativa menos costosa consiste en definir un método público que proporcione una lista ya implementada. A continuación, nos centraremos en esta segunda opción para practicar con variables cuyo tipo es el de una interfaz. Además, también se usarán las implementaciones de `Comparable` que has realizado en el ejercicio anterior.

Siguiendo esta segunda opción, se desea implementar un método en la clase `FiguresGroup` que, al aplicarlo a un grupo de figuras, devuelva un objeto que pueda ser visto como una lista de figuras ordenadas por su área y en orden creciente. Debe usarse el algoritmo de ordenación por *inserción directa*, y el perfil del método debe ser:

```
public List<Figure> orderedList()
```

Ejercicio 3 *Implementa el método* `orderedList` *en la clase* `FiguresGroup`*. Los siguientes pasos te pueden ayudar a resolver el ejercicio usando las operaciones de la interfaz* `List` *especificadas anteriormente y el comparador de la clase* `Comparable`*:*

- *Importa el paquete* `java.util` *en la clase* `FiguresGroup` *para poder usar la interfaz* `List` *y las clases* `LinkedList` *y* `ArrayList`*.*
- *Escribe el perfil del método* `orderedList`*.*
- *En el cuerpo del método se deberá crear una lista de figuras donde quedarán almacenadas, de forma ordenada, las figuras del atributo* `figuresList` *de* `this` *(figuras que están en posiciones consecutivas desde* `0` *hasta* `numF-1` *de dicho array atributo). Para ello:*
  - *Instancia una lista de figuras usando la clase* `LinkedList` *o la clase* `ArrayList` *(elige una de las dos), y asigna la referencia del objeto creado a una variable del tipo* `List`*.* De esta variable solo sabemos que podemos aplicarle los métodos definidos en `List`*, y que contendrá la referencia al objeto que devolverá el método.*
  - *Añade, si existe, el primer elemento del array* `figuresList` *a la lista de figuras. Puedes usar el método* `add(Object)` *de* `List`*.*

- *Para implementar el algoritmo de inserción directa, se requiere un recorrido y, anidado a éste, una búsqueda. Programa un* ***recorrido ascendente del array*** `figuresList` *desde la posición* `1` *hasta la posición* `numF-1`.
- *En cada iteración del recorrido, añade una figura del array en la lista ordenada, implementando una* ***búsqueda descendente de la posición*** *en la que insertar cada figura para que la lista continúe ordenada. Para ello:*
  - *Define una variable que indique la posición de la lista que se está tratando, e inicialízala al tamaño de la lista menos* 1. *Usa el método* `size()` *para obtener el tamaño de la lista.*
  - *Escribe un bucle descendente para buscar en la lista la posición en la que se debe quedar la figura que se quiere insertar. Ten en cuenta que, como mucho, se ha de llegar a la posición cero (inclusive), y que se ha de ir preguntando si la figura del array* `figuresList` *a insertar es menor que la figura visitada en la lista. Se debe usar el método* `get` *de* `List` *para obtener la figura de la lista, y el método* `compareTo` *para compararla con la figura de* `figuresList` *a insertar en la lista.*
  - *El bucle anterior se detiene en una posición anterior a la del lugar de inserción, así que habrá que insertar la figura de* `figuresList` *una posición posterior a la de parada. Se debe usar el método* `add(int, Object)` *de la interfaz para realizar la inserción en dicha posición.*

Nota*: No se debe tratar ninguna excepción dejando que se propaguen en caso de que se produzcan.*

Para comprobar el método `orderedList`, puedes usar el método `main` de una nueva clase `FiguresGroupUse`, disponible en Poliformat. En este `main`, las figuras se insertan en el grupo en el orden de las llamadas al método `add`, y después se ordenan usando el método `orderedList`.

```
public static void main(String[] args) {
  FiguresGroup g = new FiguresGroup();
  g.add(new Circle(1.0, 6.0, 6.0));
  g.add(new Rectangle(2.0, 5.0, 10.0, 12.0));
  g.add(new Triangle(3.0, 4.0, 10.0, 2.0));
  g.add(new Circle(4.0, 3.0, 1.0));
  g.add(new Triangle(5.0, 1.0, 1.0, 2.0));
  g.add(new Square(6.0, 7.0, 15));
  g.add(new Rectangle(7.0, 2.0, 1.0, 3.0));
  System.out.println(g.orderedList());
}
```

Se usa el método `println()` para mostrar por la salida estándar las figuras ordenadas por sus áreas. La lista se convierte a cadena de caracteres en un formato usado por los métodos `toString` implementados en las clases `ArrayList` y `LinkedList`, los cuales invocan los métodos `toString` de las figuras.

3. Extensión de una interfaz

Como se comentó en el citado material de lectura previa de la práctica 2, las interfaces solo pueden heredar de otras interfaces. Las derivadas de una interfaz añaden funcionalidad a las especificaciones que heredan. El hecho de implementar la interfaz derivada exige la implementación de los métodos que se heredan. Eso es muy útil cuando los nuevos métodos van a usar los métodos que se heredan como ocurre en el siguiente ejercicio.

Ejercicio 4 *Escribe una nueva interfaz con nombre* `ComparableRange` *que extienda la interfaz* `Comparable` *con el método* `int compareToRange(T o)`.

Ejercicio 5 *Haz que la clase* `Rectangle` *implemente la interfaz* `Comparable Range` *teniendo en cuenta que solo se quiere usar esta comparación para comparar pares de rectángulos y/o cuadrados.*
La norma para esta clase es que se comporte de forma similar a su clase padre excepto que considera iguales dos figuras si la diferencia de sus áreas en valor absoluto es menor o igual al 10 % de la suma de sus áreas. Si son diferentes bajo este criterio, se comparan igual que el método `compareTo` *que hereda.*

Para comprobar que la solución dada al problema anterior es correcta, se puede ampliar la clase `FiguresGroupUse` añadiendo varios rectángulos y cuadrados de lado aleatorio a una lista, y buscar los pares de figuras que son iguales bajo el nuevo criterio de igualdad. Para cada par que sea igual se puede mostrar su posición en la lista, el nombre de sus clases y su área para verificar que son iguales. Es una oportunidad para escoger un objeto del tipo `ArrayList` o `LinkedList`, eligiendo el que no hayas usado en la implementación del método `orderedList`.

El método `testComparableRange`, en la nueva clase `FiguresGroupUse`, sería una posible implementación en la que, para obtener números aleatorios se usa la clase `Random`, y `f.getClass().getName()`, para obtener el nombre de la clase de la figura `f`.

4. Diseño de una interfaz

El uso de las interfaces permite extender la funcionalidad a clases que no están necesariamente en la misma jerarquía de clases, y ni siquiera hace falta

que estén en el mismo paquete. En los siguiente ejercicios deberás crear una interfaz que implementen solo algunas de las clases de la jerarquía definida por la clase `Figure` y sus descendientes.

```
int n = (int)radius;
for (int j = 0; j <= n * 2; j++) {
    for (int i = 0; i <= n * 2; i++) {
        if (Math.pow(i - n, 2.0) + Math.pow(j - n, 2.0)
            <= (int)Math.pow(n, 2)) {
            System.out.print(c);
        }
        else {
            System.out.print(" ");
        }
    }
    System.out.println();
}
```

Figura 1: Código para dibujar círculos

```
int b = (int) base;
int h = (int) height;
for (int i = 0; i < h; i++) {
    for (int j = 0; j < b; j++) {
        System.out.print(c);
    }
    System.out.println();
}
```

Figura 2: Código para dibujar rectángulos

Se quiere añadir la funcionalidad de dibujar algunas figuras (para las que se dispone de un algoritmo que las dibuje en el terminal usando un carácter).

Ejercicio 6 *Define una interfaz de nombre* `Printable` *que especifique un método de perfil:*

```
void print(char c)
```

Las clases que implementen este método deben usar el carácter de su parámetro para dibujar sus objetos. En las figuras 1 y 2 dispones de los

algoritmos[2] para dibujar rectángulos y círculos usando su estructura de datos: `base` y `height` para los rectángulos y `radius` para los círculos. Las posiciones de dichas figuras no se consideran para dibujar en una terminal.

Ejercicio 7 *Implementa la interfaz* `Printable` *en las clases* `Circle` *y* `Rectangle`*. Aprecia que también será posible dibujar cuadrados, dado que* `Square` *heredará de* `Rectangle` *el método* `print`*.*

Se desea dibujar todas las figuras "dibujables" de un grupo de figuras, es decir, la clase `FiguresGroup` debe implementar `Printable`. Si se intenta usar el siguiente algoritmo para dibujar todas las figuras de un grupo:

```
public void print(char c) {
  for (int i = 0; i < numF; i++) {
    figuresList[i].print(c);
  }
}
```

se produce un error de compilación.

Ejercicio 8 *Implementa la interfaz* `Printable` *en la clase* `FiguresGroup`*. Debes partir de esta anterior implementación del método* `print`*, corrigiendo el código para que compile y funcione correctamente.*
Ten en cuenta que los elementos de `figuresList` *son de tipo* `Figure` *y que no todos se pueden dibujar. Así que, además de comprobar que la clase de una determinada figura del grupo implementa* `Printable`*, debes facilitar el acceso al método* `print`*, pues no está implementado en la clase* `Figure` *que es el tipo de* `figuresList`*.*
Para comprobar que funciona, puedes aplicar el método `print` *al grupo definido en la clase programa* `FiguresGroupUse`*.*

[2]Dispones, en Poliformat, de un fichero llamado `fragmentos.java` que contiene el código reproducido en estas 2 figuras.