Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20

Parte I
Programación en Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 2

Polimorfismo e interfaces en Java

Índice

1. Objetivo y planteamiento de la Práctica

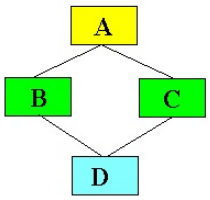
En esta segunda práctica se plantea la ampliación de la solución que se realizó en la primera práctica. La ampliación consiste en el uso de un tipo de clases en *Java*: las *interfaces*.

Antes de empezar a abordar los ejercicios de la práctica, en la sección 2, se introduce el **concepto de interfaz** junto con su utilidad, sus diferencias con las clases abstractas y su sintaxis. En la sección 3 se proponen **ejercicios** para practicar algunas de las cuestiones abordadas en la sección anterior y que debes intentar realizar planteando tus dudas a tu profesor de prácticas.

2. Introducción

2.1. ¿Qué es una interfaz?

En clases de teoría se ha estudiado la herencia como un tipo de *Polimorfismo Universal de Inclusión*. También se han estudiado las Clases Abstractas, su utilidad y uso en *Java*. En la primera práctica se realizaron ejercicios con herencia incluyendo las clases abstractas. Una clase abstracta es una clase de la que no se pueden crear objetos y puede contener métodos abstractos. Este tipo de métodos pueden verse como una serie de exigencias para las clases derivadas. El tipo de herencia usado con estas clases es simple: una clase sólo hereda de otra clase, pero puede tener muchas clases derivadas. Las interfaces introducen cierta flexibilidad en la herencia de *Java*, ya que una clase de cualquier tipo puede heredar de varias interfaces (incluidas las propias interfaces), y con ello incrementar la capacidad del polimorfismo en el lenguaje.



En algunos contextos se dice que las interfaces introducen cierto grado de *herencia múltiple*. El concepto general de este tipo de herencia consiste en que una clase puede heredar de varias clases. Esto presenta algunos problemas que se han de resolver estableciendo alguna política en el tratamiento de la herencia. Un ejemplo de esta problemática es el *problema del diamante*, el cual ocurre cuando dos clases `B` y `C` heredan un método `m` de una clase `A` y adaptan el método heredado a la clase derivada usando sobrescritura. Si otra clase `D` hereda las dos versiones del método `m` de las clases `B` y `C` ¿Cuál de las dos versiones debe usarse ante una instancia de `D`? Existen muchos lenguajes con herencia múltiple (*C++*, *Eiffel*, *Python*, *Ruby* ...), y cada uno implementa su propia política para resolver este problema.

Las interfaces son lo más parecido a la herencia múltiple que implementa *Java* aunque no resuelve los problemas inherentes a este tipo de herencia. Estas clases se asemejan a una clase abstracta pura en la que todos los méto-

dos son abstractos y públicos (sin necesidad de declararlos explícitamente como tal), y sus atributos son estáticos y finales (constantes). Esto implica que no tiene constructores y no se pueden crear objetos de una interfaz. Cuando una clase `C` hereda de una interfaz `I`, se dice que `C` ***implementa*** `I`.

2.2. ¿Para qué pueden servirnos las interfaces de Java?

Un *Tipo Abstracto de Datos* (TAD) es un conjunto de valores y operaciones que cumple con los principios de abstracción, ocultación de la información y se puede manejar sin conocer su representación interna, es decir, son independientes de la implementación. Dicho de otra forma, un TAD permite separar la especificación de una clase (qué hace) de la implementación (cómo lo hace). El uso de TAD's da lugar a programas más robustos y menos propensos a errores. Las interfaces son clases que se usan para especificar TAD's. Al implementar una interfaz, las clases extienden su funcionalidad estando obligadas ellas y/o sus derivadas a implementar los métodos abstractos heredados de la interfaz. Las implementaciones de los métodos abstractos deben usar la estructura de datos de la clase en la que se implementan. De esta forma se garantiza que a los objetos de la clase que implementa una interfaz se les pueden aplicar los métodos heredados.

Al definir interfaces también se permite la existencia de *variables polimórficas* definiéndolas con el tipo de una interfaz, y con ello también se permite la invocación polimórfica de métodos sobre estas variables. Esto restringe el uso de los objetos de una clase que implementa una interfaz a la funcionalidad especificada en la interfaz.

Otra característica de las interfaces es que nos permiten declarar *constantes* que van a estar disponibles para todas las clases que las implementen. Esto ahorra código evitando tener que escribir las mismas declaraciones de constantes en diferentes clases, a la vez que se concentran en un único lugar con la mejora que supone en el mantenimiento del código.

Una característica adicional es la *documentación* incluida en la interfaz. La propia sintaxis define qué métodos han de incluirse en una clase para cumplir con la interfaz. Se necesita documentación adicional respecto a las características comunes de las clases que la implementen y para qué servirán esos métodos. En realidad, si se implementa una interfaz, lo que se hace es ajustarse a una *norma*. Por ejemplo si se desea que los objetos de una clase se puedan comparar para ponerlos en un orden, se implementa la interfaz `Comparable` definida en el API de *Java*. Esta exige la implementación del método `compareTo()`, el perfil del método establece la norma de que el método debe devolver un valor de tipo `int` y la documentación de la interfaz establece cómo ha de funcionar: devolver 0 si son iguales y un entero negativo o positivo dependiendo de qué objeto de los dos comparados es mayor. Muchas clases predefinidas implementan esta interfaz.

Existen algunas normas aconsejables en el uso de las interfaces, entre

ellas destacamos dos:

- Respetar el *principio de segregación*: Las clases derivadas de una clase que implementa una interfaz no deberían depender de interfaces que no utiliza.
- Evitar la *contaminación de la interfaz*. Esto ocurre cuando se añade un método a una clase base simplemente porque algunas de las clases derivadas lo usan.

2.3. Diferencias entre una interfaz y una clase abstracta

Sintácticamente, una interfaz es una clase completamente abstracta, es decir, es simplemente una lista de métodos no implementados que puede incluir la declaración de constantes.[1] Una clase abstracta, además puede incluir métodos implementados y variables.

Una clase abstracta la usamos cuando deseamos definir una abstracción que englobe objetos de las distintas clases que heredan de ella. Con ello, se puede hacer uso del polimorfismo en la misma jerarquía "vertical"de clases. Una interfaz permite elegir qué clases dentro de una o diferentes jerarquías de clases incorporan una determinada funcionalidad extra. Es decir, no fuerza una relación jerárquica, simplemente permite que clases no necesariamente relacionadas puedan tener algunas características similares en su comportamiento.

2.4. Sintaxis

Definición de una interfaz. Una interfaz puede extender de varias interfaces pero de ninguna clase. Aunque los métodos que especifica son abstractos **no** se explicita la palabra `abstract` ya que por defecto todos los métodos son abstractos. Tampoco pueden ser privados ni `protected`.[2]

```
[modifVisibilidad] interface nomInterfaz [extends listaInterfaces]
   {
    [CONSTANTES]
    [ [modificador] tipoDevuelto nombreMetodo1([parámetros]);
                    ...
      [modificador] tipoDevuelto nombreMetodoN([parámetros]); ]
   }
```

Ejercicio 1 *Según la definición anterior, ¿es posible definir interfaces vacías? ¿conoces alguna?*

[1] Java 8 ya permite incluir métodos por defecto y métodos estáticos en las interfaces, pero en esta práctica suponemos el uso de una versión anterior de Java.

[2] Los corchetes indican opcionalidad y `listaInterfaces` es una lista de nombres de interfaces separados por comas.

Implementación de una interfaz. Una clase solamente puede derivar de una clase base (`extends`), pero puede implementar varias interfaces escribiendo sus nombres separados por una coma después de la palabra reservada `implements`.

```
[listaModificadores] class NomClase [extends NomClase]
                          [implements listaInterfaces]
   {
        ...
   }
```

3. Realización de la práctica

Los ejercicios que se realizan en esta práctica son continuación de la primera, y deberás añadir funcionalidades adicionales a las clases que implementaste en tres pasos: usando una interfaz predefinida, extendiendo una interfaz y diseñando una interfaz.

En teoría se ha estudiado la genericidad, y en concreto su uso en clases genéricas de *Java*. Este concepto se practicará en la próxima sesión de prácticas. En esta práctica se hace uso de clases genéricas sin indicar sus argumentos de tipo. Usadas de esta forma se las conoce como *clases "raw"*, y el compilador mostrará una advertencia de que se está haciendo un uso potencialmente peligroso de las clases ya que no podrá validar los tipos[3]. Esta característica se mantiene en las últimas versiones del lenguaje sólo para mantener la compatibilidad con versiones anteriores a *Java 5*.

3.1. Uso de una interfaz predefinida

Como se comentó en la introducción de la práctica, existe una interfaz de uso común para comparar objetos de una clase entre sí: `Comparable`. Esta interfaz especifica el método `int compareTo(Object)` (el parámetro es de tipo `Object` en su versión "raw"). 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.

[3]La advertencia ("warning") se puede eliminar añadiendo la expresión `@SuppressWarnings("unchecked")` delante del método que usa las clases genéricas de una forma "raw".

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 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.*

Una alternativa de diseño menos acertada a la propuesta en el ejercicio anterior, hubiera consistido en implementar la interfaz sólo en clases concretas. Es decir, que cada figura hubiera implementado su propio método `compareTo`. En ese caso, si se hubiera decidido que el tipo del parámetro del método fuera el de la clase donde se implementa verificándolo con `instanceof`, entonces sólo se hubieran podido comparar pares de objetos de la misma clase. Para poder comparar figuras de cualquier tipo concreto manteniendo este diseño, se hubiera tenido que usar el tipo `Figure` en el `instanceof`. Pero esto hubiera dado lugar a métodos idénticos al que has implementado en el ejercicio anterior repartidos por todas las clases concretas. 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).

En la sección 2.2 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 el 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 el 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 la 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.

Ejercicio 3 *Siguiendo la segunda de las opciones comentadas anteriormente, 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 método de ordenación por* inserción directa*, y el perfil del método es:* `public List orderedList()`*. 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`*:*

- *Añade al fichero donde se define la clase* `FiguresGroup` *la importación de la interfaz* `List` *y las clases* `LinkedList` *y* `ArrayList` *importando el paquete* `java.util`*.*
- *Escribe el perfil del método* `orderedList`*. En el cuerpo de este método se deberá crear una lista de figuras donde quedarán almacenadas, de forma ordenada, las figuras que se guardan en el atributo* `figuresList` *del grupo. Recuerda que el atributo* `numF` *indica la cantidad de figuras del grupo y que estas están guardadas en el array* `figuresList` *en las*

posiciones bajas del array hasta la posición `numF-1` *de forma consecutiva y sin posiciones vacías. El cuerpo del método lo puedes implementar de la siguiente manera:*

- *Crea una instancia de una lista usando una de las dos clases* `LinkedList` *o* `ArrayList` *(elige sólo una de las dos), y asigna la referencia del objeto creado a una variable del tipo* `List`. De esta variable sólo 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. Para ello puedes usar el método* `add(Object)` *de* `List`.
- *Para implementar el método de inserción directa, implementa un* ***recorrido ascendente del array*** `figuresList` *a partir de la posición* 1*, ya que la figura de la posición cero ya está insertada en la lista (en caso de existir). Recuerda que* `numF` *indica la cantidad de figuras guardadas en el array.*
  Inserta el resto de figuras del array en la lista ordenada implementando una ***búsqueda descendente de la posición*** *en la que debe ser insertada cada una de las figuras para que la lista continúe estando ordenada. Para insertar cada figura:*
  - *Define una nueva variable de tipo* `int` *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 hasta la posición cero (inclusive), y que se ha de ir preguntando si la figura del array* `figuresList` *que se está insertando, es menor que la figura que se está visitando en la lista. Aquí se debe usar el método* `get` *de la clase* `List` *para obtener la figura de la lista, y el método* `compareTo` *para compararla con la figura de* `figuresList` *que se está insertando 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 funcionamiento del método `orderedList` puedes definir una clase `FiguresGroupUse` donde crear un grupo de figuras con varias figuras. También puedes utilizar 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. En el siguiente método `main`, primero las figuras se insertan en el grupo en el orden de las llamadas al método `add`, y después las figuras se ordenan en el grupo utilizando el método `orderedList`.

```
public static void main(String[] a) {
    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());
}
```

3.2. Extensión de una interfaz

Como se comentó en la sección 2, las interfaces sólo 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(Object o)`.

Ejercicio 5 *Haz que la clase* `Rectangle` *implemente la interfaz* `Comparable Range` *teniendo en cuenta que sólo 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 el ejercicio 3. El código de la figura 1 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`.

```
List l = new LinkedList(); // O new ArrayList();
Random r = new Random();
for (int i = 0; i < 100; i++) {
    if (r.nextInt(2) == 0) {
        double b = r.nextDouble() * 10;
        double h = r.nextDouble() * 10;
        l.add(new Rectangle(1, 1, b, h));
    }
    else {
        double b = r.nextDouble() * 10;
        l.add(new Square(1, 1, b));
    }
}
for (int i = 0; i < 100; i++) {
    Rectangle fi = (Rectangle) l.get(i);
    for (int j = i + 1; j < 100; j++) {
        Rectangle fj = (Rectangle) l.get(j);
        if (fi.compareToRange(fj) == 0) {
            System.out.print("Figuras iguales: "
                + fi.getClass().getName() + " " + i
                + " y " + fj.getClass().getName() + " " + j
                + "\n Areas: " + fi.area()
                + ", " + fj.area() + "\n");
        }
    }
}
```

Figura 1: Código para comprobar el método `CompareToRange`

3.3. 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 sólo 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 2: 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 3: Código para dibujar rectángulos

Ejercicio 6 *Se pide añadir la funcionalidad de dibujar algunas figuras para las que se dispone de un algoritmo que las dibuje en el terminal usando un carácter. Para ello define una interfaz de nombre* `Printable` *que especifique el método* `void print(char c)`*. La clase que implemente este método debe usar el carácter de su parámetro para dibujar sus objetos. En las figuras 2 y 3 dispones de los algoritmos 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.*

Las clases que deben implementar la interfaz son las que descienden de `Figure` *excepto* `Triangle` *y* `Square`*. La primera por no disponer de algorit-*

mo y la segunda porque el método `print` *que hereda de* `Rectangle` *también sirve para dibujar cuadrados.*

Ejercicio 7 *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() {
    for (int i = 0; i < numF; i++) {
        figuresList[i].print();
    }
}
```

se produce un error de compilación. Se pide corregir el código para que compile y funcione correctamente. Ten en cuenta que los elementos de `figures List` *son de tipo Figure y que no todos se pueden dibujar. Así que además de cerciorarse de 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 correctamente puedes aplicar la instrucción `print` *al grupo definido en la clase programa* `FiguresGroupUse`*.*