Pràctiques de Llenguatges, Tecnologies i Paradigmes de programació. Curs 2019-20

Part I
Programació en Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Pràctica 2

Polimorfisme i interfícies en Java

Índex

1 Objectiu i plantejament de la Pràctica **2**

2 Introducció **2**

2.1 Què és una interfície? 2

2.2 Per a què poden servir-nos les interfícies de Java? 3

2.3 Diferències entre una interfície i una classe abstracta 4

2.4 Sintaxi 4

3 Realització de la pràctica **5**

3.1 Ús d'una interfície predefinida 5

3.2 Extensió d'una interfície 9

3.3 Disseny d'una interfície 10

1 Objectiu i plantejament de la Pràctica

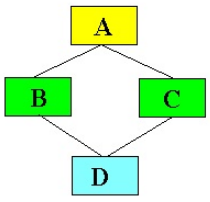
En aquesta segona pràctica es planteja l'ampliació de la solució que es va realitzar en la primera pràctica. L'ampliació consisteix en l'ús d'un tipus de classes en *Java*: les *interfícies*.

Abans de començar a abordar els exercicis de la pràctica, en la secció 2, s'introdueix el **concepte d'interfície** juntament amb la seua utilitat, les seues diferències amb les classes abstractes i la seua sintaxi. En la secció 3 es proposen **exercicis** per a practicar algunes de les qüestions abordades en la secció anterior i que has d'intentar realitzar plantejant els teus dubtes al teu professor de pràctiques.

2 Introducció

2.1 Què és una interfície?

En classes de teoria s'ha estudiat l'herència com un tipus de *Polimorfisme Universal d'Inclusió*. També s'han estudiat les Classes Abstractes, la seua utilitat i ús en *Java*. En la primera pràctica es van realitzar exercicis amb herència incloent les classes abstractes. Una classe abstracta és una classe de la qual no es poden crear objectes i pot contenir mètodes abstractes. Aquest tipus de mètodes poden veure's com una sèrie d'exigències per a les classes derivades. El tipus d'herència usat amb aquestes classes és simple: una classe només hereta d'una altra classe, però pot tenir moltes classes derivades. Les interfícies introdueixen certa flexibilitat en l'herència de *Java*, ja que una classe de qualsevol tipus pot heretar de diverses interfícies (incloses les pròpies interfícies), i amb açò incrementar la capacitat del polimorfisme en el llenguatge.



En alguns contextos es diu que les interfícies introdueixen cert grau d'herència *múltiple*. El concepte general d'aquest tipus d'herència consisteix que una classe pot heretar de diverses classes. Açò presenta alguns problemes que s'han de resoldre establint alguna política en el tractament de l'herència. Un exemple d'aquesta problemàtica és el *problema del diamant*, el qual ocorre quan dues classes `B` i `C` hereten un mètode `m` d'una classe `A` i adapten el mètode heretat a la classe derivada usant sobrescritura. Si una altra classe `D` hereta les dues versions del mètode `m` de les classes `B` i `C`, quina de les dues versions ha d'usar-se davant una instància de `D`? Existeixen molts llenguatges amb herència múltiple (*C++*, *Eiffel*, *Python*, *Ruby* ...), i cadascun implementa la seua pròpia política per a resoldre aquest problema.

Les interfícies són el més semblat a l'herència múltiple que implementa *Java* encara que no resol els problemes inherents a aquest tipus d'-

herència. Aquestes classes s'assemblen a una classe abstracta pura en la qual tots els mètodes són abstractes i públics (sense necessitat de declarar-los explícitament com a tal), i els seus atributs són estàtics i finals (constants). Açò implica que no té constructors i no es poden crear objectes d'una interfície. Quan una classe `C` hereta d'una interfície `I`, es diu que `C` ***implementa*** `I`.

2.2 Per a què poden servir-nos les interfícies de Java?

Un *Tipus Abstracte de Dades* (TAD) és un conjunt de valors i operacions que compleix amb els principis d'abstracció, ocultació de la informació i es pot manejar sense conèixer la seua representació interna, és a dir, són independents de la implementació. Dit d'una altra forma, un TAD permet separar l'especificació d'una classe (què fa) de la implementació (com ho fa). L'ús de TAD's dóna lloc a programes més robusts i menys propensos a errors. Les interfícies són classes que s'usen per a especificar TAD's. En implementar una interfície, les classes estenen la seua funcionalitat estant obligades elles i/o les seues derivades a implementar els mètodes abstractes heretats de la interfície. Les implementacions dels mètodes abstractes han d'usar l'estructura de dades de la classe en la qual s'implementen. D'aquesta forma es garanteix que als objectes de la classe que implementa una interfície se'ls poden aplicar els mètodes heretats.

En definir interfícies també es permet l'existència de *variables polimòrfiques* definint-les amb el tipus d'una interfície, i amb açò també es permet la invocació polimòrfica de mètodes sobre aquestes variables. Açò restringeix l'ús dels objectes d'una classe que implementa una interfície a la funcionalitat especificada en la interfície.

Una altra característica de les interfícies és que ens permeten declarar *constants* que van a estar disponibles per a totes les classes que les implementen. Açò estalvia codi evitant haver d'escriure les mateixes declaracions de constants en diferents classes, alhora que es concentren en un únic lloc amb la millora que suposa en el manteniment del codi.

Una característica addicional és la *documentació* inclosa en la interfície. La pròpia sintaxi defineix quins mètodes han d'incloure's en una classe per a complir amb la interfície. Es necessita documentació addicional respecte a característiques comunes de les classes que la implementen i per a què serviran els mètodes. En realitat, si s'implementa una interfície, el que es fa és ajustar-se a una *norma*. Per exemple si es desitja que els objectes d'una classe es puguen comparar per a posar-los en un ordre, s'implementa la interfície `Comparable` definida en el API de *Java*. Aquesta exigeix la implementació del mètode `compareTo()`, el perfil del mètode estableix la norma que el mètode ha de retornar un valor de tipus `int` i la documentació de la interfície estableix com ha de funcionar: retornar 0 si són iguals i un enter negatiu o positiu depenent de quin objecte dels dos comparats és

major. Moltes classes predefinides implementen aquesta interfície.

Existeixen algunes normes aconsellables en l'ús de les interfícies, entre elles destaquem dues:

- Respectar el *principi de segregació*: Les classes derivades d'una classe que implementa una interfície no haurien de dependre d'interfícies que no utilitza.
- Evitar la *contaminació de la interfície*. Açò ocorre quan s'afig un mètode a una classe base simplement perquè algunes de les classes derivades ho usen.

2.3 Diferències entre una interfície i una classe abstracta

Sintàcticament, una interfície és una classe completament abstracta, és a dir, és simplement una llista de mètodes no implementats que pot incloure la declaració de constants. [1] Una classe abstracta, a més pot incloure mètodes implementats i variables.

Una classe abstracta la usem quan desitgem definir una abstracció que englobe objectes de les diferents classes que hereten d'ella. Amb açò, es pot fer ús del polimorfisme en la mateixa jerarquia "vertical"de classes. Una interfície permet triar quines classes dins d'una o diferents jerarquies de classes incorporen una determinada funcionalitat extra. És a dir, no força una relació jeràrquica, simplement permet que classes no necessàriament relacionades puguen tenir algunes característiques similars en el seu comportament.

2.4 Sintaxi

Definició d'una interfície. Una interfície pot estendre de diverses interfícies però de cap classe. Encara que els mètodes que especifica són abstractes **no** s'explicita la paraula `abstract` ja que per defecte tots els mètodes són abstractes. Tampoc poden ser privats ni `protected`.[2]

```
[modifVisibilitat] interface nomInterficie [extends llistaInterficies]
   {
    [CONSTANTS]
    [ [modificador] tipusTornat nombreMetode1([paràmetres]);
                    ...
      [modificador] tipusTornat nombreMetodeN([paràmetres]); ]
   }
```

Exercici 1 *Segons la definició anterior, és possible definir interfícies buides? en coneixes alguna?*

[1] Java 8 ja permet incloure mètodes per defecte i mètodes estàtics en les interfícies, però en esta pràctica suposarem l'ús de una versió anterior de Java.

[2] Els claudàtors indiquen opcionalitat i `llistaInterficies` és una llista de noms d'interfícies separats per comes.

Implementació d'una interfície. Una classe solament pot derivar d'una classe base (`extends`), però pot implementar diverses interfícies escrivint els seus noms separats per una coma després de la paraula reservada `implements`.

```
[llistaModificadors] class NomClasse [extends NomClasse]
                          [implements llistaInterficies]
   {
        ...
   }
```

3 Realització de la pràctica

Els exercicis que es realitzen en aquesta pràctica són continuació de la primera, i hauràs d'afegir funcionalitats addicionals a les classes que vas implementar en tres passos: usant una interfície predefinida, estenent una interfície i dissenyant una interfície.

En teoria s'ha estudiat la genericidat, i en concret el seu ús en classes genèriques de *Java*. Aquest concepte es practicarà en la pròxima sessió de pràctiques. En aquesta pràctica es fa ús de classes genèriques sense indicar els seus arguments de tipus. Usades d'aquesta forma, es nomenen *classes "raw"*, i el compilador mostrarà un advertiment de que s'està fent un ús potencialment perillós de les classes ja que no podrà validar els tipus[3]. Aquesta característica es manté en les últimes versions del llenguatge només per a mantenir la compatibilitat amb versions anteriors a *Java 5*.

3.1 Ús d'una interfície predefinida

Com es va comentar en l'introducció de la pràctica, existeix una interfície d'ús comú per a comparar objectes d'una classe entre si: `Comparable`. Aquesta interfície especifica el mètode `int compareTo(Object)` (el paràmetre és de tipus `Object` en la seua versió "raw"). La norma estableix que el resultat d'aplicar aquest mètode a un objecte `o1` rebent com a paràmetre altre objecte `o2` (`o1.compareTo(o2)`) ha de ser:

- si `o1` = `o2` el resultat de la comparació és el valor zero.
- si `o1` < `o2` el resultat de la comparació és un valor sencer negatiu.
- si `o1` > `o2` el resultat de la comparació és un valor sencer positiu.

Aquesta norma pot concretar-se per a cada classe que la implemente. La comparació entre figures no té un ordre natural, però es pot definir un ordre

[3] L'advertiment ("warning") es pot eliminar afegint la seguent expressió davant del mètode que usa les classes genèriques d'una forma "raw": `@SuppressWarnings("unchecked")`

entre figures usant la seua àrea (grandària). Es decideix instanciar la norma anterior de la següent manera: donades dos figures `f1` i `f2` d'un tipus derivat de `Figure` la comparació

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

retorna els següents valors:

- si `f1.area()` = `f2.area()` el resultat és el valor zero.
- si `f1.area()` < `f2.area()` el resultat és un valor sencer negatiu.
- si `f1.area()` > `f2.area()` el resultat és un valor sencer positiu.

Exercici 2 *Realitza els canvis necessaris en la classe* `Figure` *de manera que puga determinar-se quan una figura és major (més gran) que una altra. Aquesta classe ha d'implementar l'interfície* `Comparable` *perquè tots els objectes de les seues classes derivades siguen comparables entre si.*

Una alternativa de disseny menys encertada que la proposta en l'exercici anterior, haguera consistit a implementar la interfície només en classes concretes. És a dir, que cada figura haguera implementat el seu propi mètode `compareTo`. En aquest cas, si s'haguera decidit que el tipus del paràmetre del mètode fora el de la classe on s'implementa verificant-ho amb `instanceof`, aleshores només s'hagueren pogut comparar parells d'objectes de la mateixa classe. Per a poder comparar figures de qualsevol tipus concret mantenint aquest disseny, s'haguera hagut d'usar el tipus `Figure` en el `instanceof`. Però açò haguera donat lloc a mètodes idèntics al que has implementat en l'exercici anterior repartits per totes les classes concretes. La solució a l'exercici que has realitzat és la desitjable en honor d'un bon disseny (més sostenible des del punt de vista del manteniment de l'aplicació).

En la secció 2.2 també es va comentar que un dels usos de les interfícies consisteix en l'especificació de TAD's. En el paquet `java.util` de *Java* està definida la interfície `List` que especifica les operacions que ha d'implementar una classe per a poder veure els seus elements com una llista. Les llistes tenen una grandària variable i els seus elements ocupen posicions numerades consecutivament amb nombres enters a partir de la posició 0. Algunes de les operacions d'aquesta interfície són:

- `void add(int index, Object element)` que insereix l'element del segon paràmetre en la posició indicada en el primer paràmetre. `IndexOutOfBoundsException` és una de les excepcions que llança aquest mètode, en concret, es llança per al cas en què la posició no existisca.
- `void add(Object element)` funciona com l'anterior però l'element s'afig al final de la llista.

- `Object get(int index)` aplicat a una llista retorna l'element que ocupa la posició indicada pel seu paràmetre. Si no existeix tal posició llança l'excepció `IndexOutOfBoundsException`.
- `int size()` retorna la quantitat d'elements de la llista.

Les classes `LinkedList` i `ArrayList` definides en el paquet `java.util.*` implementen l'interfície `List`. Cadascuna implementa les operacions anteriors tenint en compte les seues pròpies estructures de dades:

- `ArrayList`: La seua implementació es basa en un array redimensionable que duplica la seua grandària cada vegada que es necessita més espai.
- `LinkedList`: Aquesta implementació es basa en una llista doblement enllaçada on cada node té una referència a l'anterior i al següent node.

Si es volguera que els objectes de la classe `FiguresGroup` pogueren manejar-se com una llista, aquesta classe podria implementar l'interfície `List`. Açò implicaria escriure el codi de tots els mètodes com fan les classes `ArrayList` i `LinkedList`. Una altra alternativa menys costosa consisteix a definir un mètode públic que proporcione una llista ja implementada. A continuació ens centrarem en la segona opció per a practicar amb variables el tipus de les quals és el d'una interfície. A més, també s'usaran les implementacions de `Comparable` que has realitzat en l'exercici anterior.

Exercici 3 *Seguint la segona de les opcions comentades anteriorment, es desitja implementar un mètode en la classe* `FiguresGroup` *que en aplicar-ho a un grup de figures, retorne un objecte que puga ser vist com una llista de figures ordenades per la seua àrea i en ordre creixent. Ha d'usar-se el mètode d'ordenació per* inserció directa*, i el perfil del mètode és:* `public List orderedList()`*. Els següents passos et poden ajudar a resoldre l'exercici usant les operacions de la interfície* `List` *especificades anteriorment i el comparador de la classe* `Comparable`*:*

- *Afig al fitxer on es defineix la classe* `FiguresGroup` *la importació de la interfície* `List` *i les classes* `LinkedList` *i* `ArrayList` *important el paquet* `java.util`*.*
- *Escriu el perfil del mètode* `orderedList`*. En el cos d'aquest mètode s'haurà de crear una llista de figures on quedaran emmagatzemades, de forma ordenada, les figures que es guarden en l'atribut* `figuresList` *del grup. Recorda que l'atribut* `numF` *indica la quantitat de figures del grup i que aquestes estan guardades en el array* `figuresList` *en les posicions baixes del array fins a la posició* `numF-1` *de forma consecutiva i sense posicions buides. El cos del mètode pots implementar-lo de la següent manera:*

- *Crea una instància d'una llista usant una de les dues classes* `LinkedList` *o* `ArrayList` *(tria només una de les dues), i assigna la referència de l'objecte creat a una variable del tipus* `List`. D'aquesta variable només sabem que podem aplicar-li els mètodes definits en `List`, *i que contindrà la referència a l'objecte que retornarà el mètode. Afig, si existeix, el primer element del array* `figuresList` *a la llista de figures. Per a açò pots usar el mètode* `add(Object)` *de* `List`.
- *Per a implementar el mètode d'inserció directa, implementa un* ***recorregut ascendent del array*** `figuresList` *a partir de la posició* 1*, ja que la figura de la posició zero ja està inserida en la llista (en cas d'existir). Recorda que* `numF` *indica la quantitat de figures guardades en el array.*
  Insereix la resta de figures del array en la llista ordenada implementant una ***cerca descendent de la posició*** *en la qual ha de ser inserida cadascuna de les figures perquè la llista continue estant ordenada. Per a inserir cada figura:*
  - *Defineix una nova variable de tipus* `int` *que indique la posició de la llista que s'està tractant, i inicialitza-la a la grandària de la llista menys* 1. *Usa el mètode* `size()` *per a obtenir la grandària de la llista.*
  - *Escriu un bucle descendent per a cercar en la llista la posició en la qual s'ha de quedar la figura que es vol inserir. Tingues en compte que, com a molt, s'ha d'arribar fins a la posició zero (inclusivament), i que s'ha d'anar preguntant si la figura del array* `figuresList` *que s'està inserint, és menor que la figura que s'està visitant en la llista. Ací s'ha d'usar el mètode* `get` *de la classe* `List` *per a obtenir la figura de la llista, i el mètode* `compareTo` *per a comparar-la amb la figura de* `figuresList` *que s'està inserint en la llista.*
  - *El bucle anterior es deté en una posició anterior a la del lloc d'inserció, així que caldrà inserir la figura de* `figuresList` *una posició posterior a la de parada. S'ha d'usar el mètode* `add(int, Object)` *de la interfície per a realitzar la inserció en aquesta posició.*

Nota*: No s'ha de tractar cap excepció deixant que es propaguen en cas que es produïsquen.*

Per a comprovar el funcionament del mètode `orderedList` pots definir una classe `FiguresGroupUse` on crear un grup de figures amb diverses figures. També pots utilitzar el mètode `println()` per a mostrar per l'eixida estàndard les figures ordenades per les seues àrees. La llista es converteix

a cadena de caràcters en un format usat pels mètodes `toString` implementats en les classes `ArrayList` i `LinkedList`, els quals invoquen els mètodes `toString` de les figures. En el següent mètode `main`, primer les figures s'insereixen en el grup en l'ordre de les trucades al mètode `add`, i després les figures s'ordenen en el grup amb el mètode `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 Extensió d'una interfície

Com es va comentar en la secció 2, les interfícies només poden heretar d'altres interfícies. Les derivades d'una interfície afigen funcionalitat a les especificacions que hereten. El fet d'implementar la interfície derivada exigeix la implementació dels mètodes que s'hereten. Açò és molt útil quan els nous mètodes van a usar els mètodes que s'hereten com ocorre en el següent exercici.

Exercici 4 *Escriu una nova interfície amb nom* `ComparableRange` *que estenga la interfície* `Comparable` *amb el mètode* `int compareToRange(Object o);`.

Exercici 5 *Fes que la classe* `Rectangle` *implemente la interfície* `ComparableRange` *tenint en compte que només es vol usar aquesta comparació per a comparar parells de rectangles i/o quadrats. La norma per a aquesta classe és que el mètode* `compareToRange` *haurà de retornar els mateixos valors que* `compareTo` *(negatiu, zero, positiu) però tenint en compte que dos rectangles (i/o quadrats) són iguals si la diferència de les seues àrees en valor absolut és menor o igual al 10% de la suma de les seues àrees.*

Per a comprovar que la solució donada al problema anterior és correcta, es pot ampliar la classe `FiguresGroupUse` afegint diversos rectangles i quadrats de costat aleatori a una llista, i cercar els parells de figures que són iguals sota el nou criteri d'igualtat. Per a cada parell que siga igual es pot mostrar la seua posició en la llista, el nom de les seues classes i la seua àrea per a verificar que són iguals. És una oportunitat per a escollir un objecte

del tipus `ArrayList` o `LinkedList` triant el que no hages usat en l'exercici 3. El codi de la figura 1 seria una possible implementació en la qual per a obtenir nombres aleatoris s'usa la classe `Random` i `f.getClass().getName()` per a obtenir el nom de la classe 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: Codi per a comprovar el mètode `compareToRange`

3.3 Disseny d'una interfície

L'ús de les interfícies permet estendre la funcionalitat a classes que no estan necessàriament en la mateixa jerarquia de classes, i ni tan sols fa falta que estiguen en el mateix paquet. En els següent exercicis hauràs de crear una interfície que implementen només algunes de les classes de la jerarquia definida per la classe `Figure` i els seus descendents.

Exercici 6 *Es demana afegir la funcionalitat de dibuixar algunes figures per a les quals es disposa d'un algorisme que les dibuixe en el terminal usant un*

```
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: Codi per a dibuixar cercles

```
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: Codi per a dibuixar rectangles

caràcter. Per a açò defineix una interfície de nom `Printable` *que especifique el mètode* `void print(char c)`*. La classe que implemente aquest mètode ha d'usar el caràcter del seu paràmetre per a dibuixar els seus objectes. En les figures 2 i 3 disposes dels algorismes per a dibuixar rectangles i cercles usant la seua estructura de dades:* `base` *i* `height` *per als rectangles i* `radius` *per als cercles. Les posicions d'aquestes figures no es consideren per a dibuixar en una terminal.*

Les classes que han d'implementar la interfície són les que descendeixen de `Figure` *excepte* `Triangle` *i* `Square`*. La primera per no disposar d'algorisme i la segona perquè el mètode* `print` *que hereta de* `Rectangle` *també serveix per a dibuixar quadrats.*

Exercici 7 *Es desitja dibuixar totes les figures "dibuxables" d'un grup de figures, és a dir, la classe* `FiguresGroup` *ha d'implementar* `Printable`*. Si s'intenta usar el següent algorisme per a dibuixar totes les figures d'un grup:*

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

es produeix un error de compilació. Es demana corregir el codi perquè compile i funcione correctament. Tingues en compte que els elements de `figures List` *són de tipus Figure i que no tots es poden dibuixar. Així que a més de cerciorar-se que la classe d'una determinada figura del grup implementa* `Printable`*, has de facilitar l'accés al mètode* `print`*, doncs no està implementat en la classe* `Figure` *que és el tipus de* `figuresList`*.*
Per a comprovar que funciona correctament pots aplicar la instrucció `print` *al grup definit en la classe programa* `FiguresGroupUse` *.*