Université de Montpellier / Faculté Des Sciences

Résumé de cours 1
Types de données abstraits et structures de données
Interfaces en Java
L'interface `List`

Structures de données — HMIN 215

16 janvier 2019

1 Présentation du module

Dans ce module, nous nous attacherons à :
- comprendre les notions de type abstrait de données et de structure de données,
- utiliser des structures de données,
- implémenter quelques structures de données classiques (liste, pile, arbre, graphe),
- consolider notre connaissance de Java, en particulier sur tous les aspects permettant une mise en œuvre des structures de données (types interfaces, classes internes, généricité, assertions, exceptions, itérateurs, streams et collectors).

2 Type abstrait

Nous avons vu qu'un programme manipule des informations (des données) :
- simples, telles que les entiers, les booléens, les caractères.
- plus ou moins structurées et complexes comme les objets, les tableaux, les vecteurs/arrays lists.

Pour *spécifier*, s'abstraire de l'implémentation et mettre en évidence les concepts relatifs à des ensembles de données, on définit la notion de *Type abstrait*. Un *Type abstrait* comporte :
- le type défini (l'explicitation de son nom),
- les autres types utilisés,
- les opérations,
- les préconditions,
- les axiomes.

Par exemple, un type abstrait *Liste de chaînes de caractères* serait (partiellement) décrit par les éléments suivants (qui peuvent être décrits formellement, mais nous ne rentrerons pas dans le formalisme dans le cadre de ce cours) :
- le type défini : **Liste**
- les autres types utilisés : **Chaîne de caractères**
- les opérations : par exemple, ajouter une certaine chaîne de caractères, retirer une certaine chaîne de caractères, obtenir la chaîne de caractères en position i.

— les préconditions : par exemple on ne peut retirer une chaîne de caractères que si la liste n'est pas vide
— les axiomes : par exemple, après ajout d'une chaîne de caractères c dans la liste, la liste contient c.

3 Structure de données

Une structure de données est une **organisation logique** d'un ensemble de données, d'informations. On peut la comprendre comme une étape de conception avant le codage dans un langage de programmation. Par exemple, une liste de chaînes de caractères peut être organisée de manière logique dans un tableau, un arbre, une structure séquentielle de cellules chaînées entre elles. On peut également décider de trier les chaînes ou non. Cette organisation logique aura des propriétés particulières en termes de coût : place occupée pour stocker et organiser l'information, temps théorique d'exécution des opérations. Nous reviendrons sur ces aspects plus tard dans le cours.

Une structure de données vient donc offrir une mise en œuvre d'un type abstrait. Un type abstrait aura souvent plusieurs mises en œuvre. Dans les documents que vous pourrez lire sur différentes sources, vous observerez que l'on fait souvent la confusion entre type abstrait de données et structure de données, car il y a une certaine influence mutuelle entre l'organisation logique et les opérations proposées.

4 Interface Java

Le langage Java introduit la notion d'*interface* pour traduire une partie de la définition d'un type abstrait, plus exactement :
— le type défini,
— les types utilisés,
— les opérations et une partie de leurs préconditions.

On dit également qu'une *interface* définit un *contrat* (ou cahier des charges) pour un ensemble de classes. Par contrat on entend un ensemble d'opérations, incluant des opérations d'accès à des données, que l'on s'attend à avoir dans toutes les classes respectant ce contrat.

Une interface Java regroupe plus exactement :
— Dans les versions de Java $\leq$ Java 1.7
  — des méthodes d'instance publiques abstraites (avec les modifieurs `public abstract`),
  — des attributs de classe constant publics (avec les modifieurs `public final static` et une valeur).
— A partir de Java 1.8
  — des méthodes d'instance publiques présentant des comportements par défaut (avec les modifieurs `public` et `default`),
  — des méthodes statiques,
  — des types internes (développés dans un prochain cours).

On remarquera qu'une interface ne contient donc jamais ni constructeur, ni le code des méthodes (en dehors des méthodes par défaut et des méthodes statiques), ni aucun attribut d'instance.

4.1 Définition d'une interface

Le code ci-dessous montre une interface (très simplifiée) décrivant les quadrilatères. Un quadrilatère a 4 côtés et dispose de deux opérations permettant de connaître respectivement son périmètre et sa surface.

```
public interface Iquadrilatere {
   public static final int nbCotes = 4;
   public abstract double perimetre();
   public abstract double surface();
}
```

Les groupes de modifieurs suivants peuvent être omis :
— public static final
— public abstract
— public
On écrira donc le plus souvent une interface comme suit :

```
public interface Iquadrilatere {
   int nbCotes = 4;
   double perimetre();
   double surface();
}
```

Question 1 *Ecrire une interface pour une liste de chaînes de caractères avec des opérations d'ajout d'élément, pour connaître la taille, pour connaître si un élément appartient à la liste, et pour récupérer un élément appartenant à la liste et rangé à un certain indice i.*

Réponse 1

```
public interface ListeChaineCaracteres
{
    void add(String element);
    boolean contains(String o);
    String  get(int index);
    int size();
}
```

4.2 Spécialisation d'une interface

De même que l'on peut spécialiser des classes, on peut également spécialiser des interfaces. Par exemple, l'interface `Iquadrilatere` peut être spécialisée par l'interface `Irectangle`. Le mot-clef `extends`, déjà rencontré pour indiquer qu'une classe en spécialise une autre, est utilisé ici aussi.

Dans cette interface, on précise que l'angle est toujours de 90^0, et quatre opérations permettent de connaître ou de modifier respectivement la largeur et la hauteur.

De plus, des implémentations **par défaut** sont données pour les méthodes de calcul du périmètre et de la surface, ainsi qu'une description sous forme d'une chaîne de caractères. Ces méthodes peuvent être écrites à ce stade puisqu'elles appellent des méthodes dont l'existence est assurée par le contrat (les méthodes d'instance publiques prévues). Elles

sont introduites par le mot-clef `default`. En Java, il y a une restriction sur les méthodes `default` importante à connaître. Les méthodes apparaissant également dans la classe `Object` comme la méthode `toString` par exemple, ne peuvent pas être des méthodes `default` d'une interface.

Enfin, un exemple de méthode `static` est donné. Cette méthode vérifie l'égalité de deux rectangles (donnée par celle de leurs largeurs et de leurs hauteurs respectives).

```
public interface Irectangle extends Iquadrilatere{
    int angle = 90;
    double getLargeur();
    void setLargeur(double l);
    double getHauteur();
    void setHauteur(double h);

    default double perimetre()
    {return 2*this.getLargeur()+2*this.getHauteur();}

    default double surface()
    {return this.getLargeur()*this.getHauteur();}

    default String description() // on ne peut pas utiliser le nom "toString"
    {return "largeur ="+this.getLargeur()+" hauteur ="+this.getHauteur();}

    static boolean egal(Irectangle r1, Irectangle r2)
    {
        return r1.getLargeur()==r2.getLargeur()
            && r1.getHauteur()==r2.getHauteur();
    }
}
```

Au contraire d'une classe qui ne peut avoir qu'une super-classe directe, une interface peut avoir plusieurs super-interfaces directes. On parle de spécialisation **multiple**.

Par exemple, on peut définir une interface représentant les objets colorés puis une interface spécialisant les rectangles et les objets colorés. Elle étend directement deux autres interfaces et introduit une nouvelle méthode `repeindre`.

```
import java.awt.Color;

public interface IobjetColore {
    Color couleurDefaut = Color.white;
    Color getCouleur();
}

public interface IrectangleColore extends IobjetColore, Irectangle{
    void repeindre(Color c);
}
```

En cas d'héritage de deux méthodes par défaut dans une classe ou dans une interface, une erreur se produit, il faut alors proposer une solution (une nouvelle implémentation de cette méthode) sur le lieu de l'erreur.

4.3 Implémentation d'une interface

Une interface s'implémente à l'aide d'une classe. Par exemple nous proposons dans le code ci-dessous une classe `RectangleAT` qui implémente les méthodes des interfaces qu'elle spécialise (dont elle respecte le contrat).

`RectangleAT` implémente directement `Irectangle` (notez le mot-clef **implements**) et indirectement `Iquadrilatere` qui est une généralisation de `Irectangle`. Cette implémentation stocke la largeur et la hauteur dans deux attributs.

```
public class RectangleAT implements Irectangle {

    private double largeur, hauteur;

    public RectangleAT() {}

    public RectangleAT(double largeur, double hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    @Override
    public double getLargeur() {return this.largeur;}

    @Override
    public void setLargeur(double l) {this.largeur = l;}

    @Override
    public double getHauteur() {return this.hauteur;}

    @Override
    public void setHauteur(double h) {this.hauteur = h;}
}
```

5 La place des interfaces dans l'API de Java

L'API de Java contient de nombreuses interfaces.
Certaines décrivent des types abstraits de données comme :

- L'interface `List`
  - qui décrit les séquences d'éléments et offre des méthodes telles que :
    - insertion `add(valeur)`, récupération `get(int i)`, taille `size()`, etc.
  - qui est implémentée par des classes telles que `Vector`, `ArrayList`, `LinkedList`
- L'interface `Map`
  - qui décrit les dictionnaires associatifs et offre des méthodes telles que :
    - insertion `put(clef, valeur)`, récupération `get(clef)`, taille `size()`, etc.
  - qui est implémentée par des classes telles que `Hashtable`, `HashMap`, `TreeMap`
- L'interface `Comparable`
  - Comprenant la méthode `int compareTo(autreObjet)`
  - Une séquence contenant des éléments respectant (c'est-à-dire instances d'une classe implémentant) l'interface `Comparable` pourra être triée grâce à des méthodes prédéfinies dans l'API comme `Collections.sort`.

D'autres sont vides de méthodes, on dit que ce sont des interfaces *marqueurs* indiquant si un objet peut être soumis à un certain traitement :

— `Cloneable` : pour les objets qui seront munis d'une méthode `clone` publique par redéfinition de la méthode `protected Object clone()` de la classe `Object`.

— `Serializable` : pour les objets que l'on pourra sérialiser, c'est-à-dire placer dans des flux d'objets (et donc concrètement dans des fichiers avec un format particulier).

6 Synthèse

Pour synthétiser, les interfaces sont :

— des types plus abstraits que les classes, plus largement réutilisables,

— une technique pour masquer l'implémentation en définissant une partie d'un type abstrait, qui favorise l'écriture de code plus général (ce sera vu plus en détails en TD),

— des types qui améliorent la structuration par spécialisation (ou implémentation) multiple : entre interfaces, entre classes et interfaces.

TABLE 1 – Comparaison entre les différents types construits de Java (autres que les tableaux et enums)

Type	Classe	Classe abstraite	Interface
attribut de classe `static public final`	x	x	x
attribut d'instance	x	x	
attribut de classe `static` non public ou non final	x	x	
méthode `public abstract`	x	x	x
méthode `public default`			x
méthode `d'instance` sans `default`	x	x	
méthode `public static`	x	x	x
méthode non `public`, ou non `static`, ou non `default`	x	x	
constructeur	x	x	