h e g
Haute école de gestion
Genève

Java - suite

63-31 - Programmation Collaborative

Claude Chaudet

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Résumé des épisodes précédents

Programmation et Conception Orientées Objets

- Classes & Objets
- Attributs & méthodes ; Encapsulation
- Attributs & méthodes statiques
- Héritage
- Classes abstraites, interfaces, polymorphisme

Java

- Readers
- Exceptions
- Comparables
- Collections

  Listes (Vector, ArrayList, LinkedList), Ensembles (Set), Dictionnaires (Map), Files (Queue, Dequeue)

Types Abstraits de Données (TAD)

Définition

Un type abstrait un type défini par son interface, i.e. l'ensemble des méthodes permettant d'y accéder

- On définit ce que va faire la classe, pas comment elle va le faire
- Notion de contrat pour s'affranchir des contraintes (raisonnablement)
- Interface claire pour l'utilisateur du type

Exemples courants :

- Tableau
- Liste
- Dictionnaire (tableau associatif / map)
- Ensemble
- File / Pile
- ...

Formalisme

Lorsqu'on définit un type abstrait de données, on spécifie 3 éléments

Des opérations définies comme des constantes ou des primitives (fonctions mathématiques) (ensemble de départ & d'arrivée) dont découleront les signatures des méthodes

ListeVide : $\emptyset \rightarrow$ Liste(T)

Taille : Liste(T) $\rightarrow \mathbb{N}$

Ajouter : T $\times$ Liste(T) $\rightarrow$ Liste(T)

Retirer : Liste(T) $\rightarrow$ T $\times$ Liste(T)

Des conditions de validité des opérations

Retirer valide si Taille $>$ 0

Des axiomes, i.e. des propriétés (sémantique) qui sont tout le temps vraies ; définissent le comportement

Taille(ListeVide()) = 0

Taille(Ajouter(x, L)) $>$ 0

Retirer(Ajouter(x, L)) = (x, L)

Raisons d'être

Le formalisme est une spécification algébrique

- On peut prouver des choses dessus, indépendamment de l'implémentation

Permet de convenir d'un ensemble de fonctions avec des non-informaticiens

Permet de définir un contrat de vérification de l'implémentation

- Ecrire des jeux de tests pour chacune des propriétés identifiées

Architecture plus modulaire : on utilise des TAD plutôt que des types concrets dans le code pour être indépendants de l'application

Permet de changer l'implémentation pour s'adapter à la plate-forme d'exécution ou en fonction du scénario

- CPU vs. GPU, architecture matérielle, questions de performance, de mémoire, etc.

Exemple / Exercice

Implémenter une classe Date :

✦ Date()	Constructeur d'une date vide
✦ Date(? jj, ? mm, ? aa)	Constructeur avec valeurs
✦ int Jour()	Donne le jour
✦ int Mois()	Donne le mois
✦ int Annee()	Donne l'année

Plusieurs versions possibles:

- ✦ Année sur 2 ou 4 chiffres
- ✦ Mois numérique ou 2 lettres (Ja, Fe, Ma, ...)
- ✦ Jour de l'année numérique (1 .. 366) et année

En pratique

Supposons qu'on a une méthode qui compte le nombre d'éléments dans une collection d'entiers

```
public static int CountItems(ArrayList<Integer> myArray)
{
        int n = 0;

        for (Integer x: myArray)
                n++;

        return n;
}

public static void main(String[] args)
{
        ArrayList<Integer> al = new ArrayList<>();

        al.add(13);
        al.add(67);

        System.out.println("Nombre d'éléments : " + CountItems(al));
}
```

Comment l'adapter à d'autres types de collections ?

```
public static int CountItems(Vector<Integer> myVector) { … }
public static int CountItems(Vector<Integer> LinkedList) { … }
```

Utilité des TAD

Tous les types qui nous intéressent implémentent l'interface collection qui implémente Iterable. On n'a pas besoin de plus qu'iterable ici.

```
public static int CountItems(Iterable<Integer> myIterable)
{
                int n = 0;

                for (Integer x: myIterable)
                                n++;

                return n;
}

public static void main(String[] args)
{
                ArrayList<Integer> al = new   ArrayList<>();
                Vector<Integer> v = new       Vector<>();

                al.add(13);
                al.add(67);
                v.add(123);

                System.out.println("Hello " + CountItems(al));
                System.out.println("Hello " + CountItems(v));
}
```

Enumérations

Les énumérations en Java

Une énumération est une liste de valeurs restreintes

- Exemple : {Bleu, Rouge, Jaune, Orange, Vert}

Déclaration

```
public enum Couleur {BLEU, ROUGE, JAUNE, ORANGE, VERT}
```

```
class TestEnum {
            enum Couleur {BLEU, ROUGE, JAUNE, ORANGE, VERT}

            public static void main(String[] args)
            {
                        System.out.println("Couleur du ciel : " + Couleur.BLEU);
            }
}
```

Classes énumérations

Une énumération est une classe qui dérive de la classe Enum

Méthodes notables :

toString()	=> Renvoie une chaîne de caractères qui correspond à la valeur de l'élément
valueOf()	=> Renvoie la valeur de l'énumération à partir d'une chaîne de caractères
values()	=> Liste les valeurs possibles de l'énumération (tableau)
ordinal()	=> Donne l'index (démarre à 0) d'une valeur dans une énumération
compareTo()	=> comparaison sur la base des numéros d'index

```
class TestEnum {
		enum Couleur {BLEU, ROUGE, JAUNE, ORANGE, VERT}

		public static void main(String[] args)
		{
			System.out.println("Couleur du ciel : " + Couleur.BLEU);
			System.out.println("Valeur de VERT : " + Couleur.valueOf("VERT"));
			System.out.println("Index de JAUNE : " + Couleur.JAUNE.ordinal());
		}
}
```

Améliorer les énumérations

On peut faire une "vraie" classe énumération (dans un fichier séparé) et définir un constructeur privé afin d'associer des valeurs aux éléments

```
public enum Couleur {
              BLEU("#0084D1"), ROUGE("#C5000B"), JAUNE("#FFD320"), ORANGE("#FF8000"), VERT("#008000")

              private String codeRGB;

              private Couleur(String rgb) {
                            this.codeRGB = rgb;
              }

              public String RGB() {
                            return this.codeRGB;
              }

}
```

Exercice

1) Tester la classe couleur

2) Surcharger la méthode toString() afin d'afficher chaque couleur comme suit:

BLEU (#0084D1)

3) Ajouter un attribut privé "valeur" qui prend les valeurs suivantes:

JAUNE = 1 ; BLEU = 2 ; ROUGE = 3 ; ORANGE = 4 ; VERT = 5

4) Peut-on utiliser cet attribut afin de définir notre propre ordre des couleurs en surchargeant la méthode compareTo()?

Essayer de le mettre en place ; que se passe-t-il ?

Que peut-on en conclure ?

Classes/types génériques

Raison d'être

Java est fortement typé => vérification de la compatibilité des types au moment de la compilation

Comment alors utiliser le même code sur plusieurs types d'objets ?

- Exemple : créer une liste chaînée d'objets

On pourrait dupliquer du code

- => Maintenance est difficile et incertaine

On pourrait utiliser le type générique Object (ou un type dérivé) et profiter du polymorphisme

- Transtypages (cast) fréquents
  Opération coûteuse en temps

- Vérification de la compatibilité des types à l'exécution
  Plantages et levée d'exceptions.

Définition

Une classe générique (ou simplement "générique") est une classe qui a une (ou plusieurs) autre(s) classe(s) en paramètre

On décide du typage effectif au moment de la compilation

Exemple : ArrayList‹E› –› E peut être Integer, String ou MaClasse sans changer l'implémentation

Définition:

```
public class MaClasseGenerique <ClasseDeBase>
{
        ClasseDeBase attr1;
        …
        public MaMethode(ClasseDeBase x)
        {
                …
        }

        …
}
```

Utilisation:

```
MaClasseGenerique<Integer> = new MaClasseGenerique<Integer>();
```

Types génériques : à noter

On peut utiliser plusieurs classes paramètres :

```
public class MaClasseGenerique <ClasseDeBase, AutreClasse, EncoreUne>
```

Utilisation d'interfaces et héritage

```
public class MaClasseGenerique <A, B> extends MaClasse<A> implements MonInterface<A,B>
```

Restriction à des types qui héritent d'une classe ou d'une ou plusieurs interface(s)

```
public class MaClasseGenerique <T extends MonTypePere>
public class MaClasseGenerique <T extends MonInterface>
public class MaClasseGenerique <T extends MonInterface1 & MonInterface2>
```

Attention

Supposons qu'on a deux classes A et B telles que B hérite de A (B extends A)

Est-ce que MaClasseGenerique<B> hérite de MaClasseGenerique<A>?

> En d'autres termes, est-ce que je peux définir une fonction qui prenne en paramètre MaClasseGenerique<A> et accepte les deux ?

Attention

Supposons qu'on a deux classes A et B telles que B hérite de A (B extends A)

Est-ce que MaClasseGenerique‹B› hérite de MaClasseGenerique‹A›?

> En d'autres termes, est-ce que je peux définir une fonction qui prenne en paramètre MaClasseGenerique‹A› et accepte les deux ?

==› Non

Mais Java permet de spécifier le paramètre comme "toute classe étendant A" avec un joker:

> ‹? extends A›

Méthodes génériques

On peut déclarer, dans une classe normale, une méthode générique

```
public  class MaClasse {

    public  static <T> T maMethode(T[] tableaudeT) { ... }
}
```

- La déclaration du type générique se fait avant le type de retour de la méthode
  Permet de l'utiliser comme type de retour

Appel :

```
MaClasse.<String>maMethode("Premiere Chaine", "Deuxieme Chaîne");
```

Exercice

Ecrire une classe générique Triplet‹E› qui prend 3 valeurs d'un même type E

- Constructeur(s)
- Accès à chaque élément : Premier(), Deuxieme() et Troisieme()
- Affichage (toString)

L'instancier sur plusieurs types de base et tester.