Programmation Java TP #4

A. Pigeau

L'objectif de ce TP est de découvrir l'API Java, en particulier, les classes d'entrées/sorties, le clonage et les génériques.

1 Exercice : [In & Out Compréhension]

L'objectif de cet exercice est de comprendre comment fonctionne la bibliothèque d'entrée/sortie de `Java`. Nous vous proposons de partir d'un exemple plus simple : le `PolyHero`. Une instance d'un `PolyHero` est un super Héro avec un costume configurable (short, caleçon, cape, ceinture, casque, casquette, botte,...).

1. créer un package `nomEtudiant.tp4.hero`
2. implémenter la classe `PolyHero`, celle-ci contiendra simplement le nom du personnage
3. implémenter une classe `PolyHeroDeco` contenant simplement la méthode `description` abstraite. Cette méthode retourne une description de l'accessoire sous forme de chaine de caractère ("casque" pour la classe `Casque` par exemple)
4. implémenter vos différents accessoires, héritant de `PolyHeroDeco`
5. votre objectif est maintenant de pouvoir ajouter des accessoires à votre héro. Voici notre solution :
   - ajouter une classe `Hero`, père des classes `PolyHero` et `PolyHeroDeco`. Cette classe contient une méthode `description` abstraite. Vous devez instancier la méthode `description` dans les sous classes. Pour un héro, elle retournera juste son nom sous forme de chaine de caractères.
   - ajouter un attribut `myHero` de type `Hero` dans la classe `PolyHeroDeco`
   - ajouter un constructeur à `PolyHeroDeco` prenant en paramètre un `Hero`. L'attribut `myHero` est initialisé avec ce paramètre. La classe `PolyHeroDeco` et ses fils ne doivent pas avoir d'autres constructeurs.
   - la méthode `description` des accessoires doit maintenant retourner la concaténation de la description de son attribut `myHero` et de son accessoire

   Une autre solution aurait été d'ajouter un tableau d'accessoires à la classe `Hero`. Cette solution est aussi correcte mais notre objectif ici de comprendre les entrées/sorties de Java.
6. notre modélisation est maintenant complète. Il ne nous reste plus qu'à instancier notre `PolyHero` et à l'habiller :

```
Hero poly = new PolyHero("Bob");
PolyHeroDeco casque = new Casque(poly);
PolyHeroDeco ceinture = new Ceinture(casque);

System.out.println(ceinture.description()); // affichage de "Bob : casque, ceinture"
```

7. ajoutez de nouveaux accessoires pour vos héros. Combien de classes ont été modifiées ?

2 Exercice : [In & Out]

Maintenant que vous avez compris cette modélisation, vous pouvez vous exercer à utiliser les classes d'entrées/sorties de Java :

- vos nouveaux héros : `FileInputStream`, `PipedInputStream`, `ObjectInputStream`
- vos nouveaux accessoires :

- `DataInputStream` : offre la possibilité de lire directement des types primitifs (double, char, int), ceci grâce à des méthodes comme readDouble() , readInt() ...
- `BufferedInputStream` : cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon et, dès que celui-ci est rempli, le programme a accès aux données ;
- `PushbackInputStream` : permet de remettre un octet déjà lu dans le flux entrant ;
- `LineNumberInputStream` : cette classe offre la possibilité de récupérer le numéro de ligne lue à un instant T.

Tester les classes `FileInputStream` et le filtre `BufferedInputStream` :

1.
2. créer un package `nomEtudiant.tp4.inOut`
3. Récupérer une image sur le web et ajouter la à votre projet Eclipse (aller dans votre workspace et ajouter le fichier dans la racine de votre projet. Dans Eclipse, cliquez droit dans votre projet et cliquez sur `Refresh`)
4. créer une classe `TestInOut` avec simplement une méthode `main`
5. implémenter cette méthode en ouvrant tout d'abord votre fichier avec un `FileInputStream` et en lisant tout les octets un à un. Vous utiliserez la méthode `java.lang.System.currentTimeMillis()` pour calculer le temps mis à lire le fichier
6. implémenter le même test en utilisant maintenant un filtre `BufferedInputStream`. Comparez les performances des deux approches.

3 Exercice : [Ensemble & clone]

L'objectif de cet exercice est d'utiliser les collections de `Java` et de maîtriser la copie d'objets.

1. créer un package `nomEtudiant.tp4.personne`
2. implémenter une classe `Personne`. Chaque personne a un nom, un prénom, une liste d'enfants, un genre (masculin ou féminin) et un numéro de sécurité sociale. Le genre doit être implémenté à l'aide d'une énumération ;
3. implémenter une classe `Secu` contenant un ensemble de `Personne`. Sa méthode `toString` doit afficher l'ensemble des informations sur les personnes qu'elle contient. Utiliser un itérateur pour parcourir l'ensemble ;
4. implémenter une classe interne à `Secu` pour tester votre classe. Vérifier que votre ensemble ne peut pas contenir de personne avec un numéro de sécurité sociale identique ;
5. proposer une méthode pour créer des copies de `Personne`. Il ne faut pas oublier de faire une copie de ses enfants ;
6. proposer une méthode pour créer une copie d'une instance de `Secu` ;
7. proposer une méthode pour dé/sérialiser l'ensemble des personnes (voir la bibliothèque `java.io.*`)
8. proposer une méthode pour sauvegarder la liste des personnes dans un fichier texte comprenant les trois colonnes suivantes :
   - Nom
   - Prénom
   - Numéro de sécurité sociale

4 Exercice : [Classe générique]

L'intérêt des génériques en Java est plus limité qu'en C++. L'avantage porte essentiellement sur les conteneurs, afin de garantir que l'ensemble des objets contenus ait le même type (l'utilisateur n'a ainsi plus besoin de *caster* les éléments quand il les récupère dans le conteneur).

En pratique, Java compile les classes génériques en enlevant tous les types paramétriques : ils sont remplacés par le type le plus général, `Object`, si aucune contrainte d'héritage n'est mise sur le type paramétrique. Par exemple : `Vector<String>` est remplacé par `Vector` et `T objetA` est remplacé par `Object objetA`. Quand le résultat de cette compilation ne convient pas, Java rajoute automatiquement les *Cast* nécessaires. La compilation de la classe générique donne ainsi lieu à une seule classe.

Exemple : le code tapé

```
public String elementListe(String x) {
 List<String> a = new LinkedList<String>();
 a.add(x);
 return a.iterator().next();
}
```

Et sa compilation :

```
public String elementListe(String x) {
 List a = new LinkedList; // plus de type generique
 a.add(x);
 return (String) a.iterator().next();
}
```

Cette méthode permet de garder une compatibilité entre des applications `Java` utilisant des génériques et des bibliothèques développées avant les génériques. Les possibilités ne sont donc pas aussi large qu'en C++ [1].

Les conséquences de cette approche sont les suivantes :

- nous ne pouvons pas utiliser les génériques sur des types primitifs ;
- nous ne pouvons pas instancier notre type générique (`T objet = new T()` provoque une erreur). En effet pour instancier l'objet nous devons appeler son constructeur, or nous ne connaissons pas son type à la compilation ;
- nous ne pouvons pas utiliser les types génériques pour des variables `static` ou dans des méthodes `static`, ces attributs ou méthodes `static` étant communs à toutes les instances (un problème se pose si j'instancie deux classes `Pile<String>` et `Pile<Integer>` et que ma classe pile `Pile<T>` contient un attribut `static` de type `T`.

Les questions suivantes mettent en valeur les avantages et inconvénients des génériques en Java.

1. créer un package `nomEtudiant.tp4.stackGen`
2. Modifier la classe `VectorStack` du TP précédent pour la rendre générique. L'utilisateur définira ainsi le type du contenu de la pile au moment de l'instanciation de la classe.
3. Définir une classe `FuncNumber` contenant un attribut n de type générique héritant de `Number` (**attention : c'est le type de l'attribut qui hérite de** *Number*) et permettant de calculer les fonctions $n!$, $fibo(n)$ (Fibonacci) et $\sum_{i=1}^{n} n$. Le type générique est donc limité en pratique à `Integer`, `Short` ou `Long`. Implémenter la classe `FuncNumber`.

Rappel pour Fibonacci :

$$F(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases} \quad (1)$$

5 Exercice Bonus : [Modélisation]

L'entreprise *PolyEntreprise* a l'organisation suivante :

- un employé est défini par :
  - un nom

1. Extrait de la documentation Java sur le site de Sun : *If you are familiar with C++'s template mechanism, you might think that generics are similar, but the similarity is superficial. Generics do not generate a new class for each specialization, nor do they permit "template metaprogramming."*

  - un prénom
  - un âge
- un employé de l'entreprise est soit un développeur, soit un manager
- un manager est responsable d'un ensemble d'employés
- un employé a généralement un manager qui le dirige (seul le manager en chef, dirigeant de l'entreprise, n'a pas de manager)

Question :

1. créer un package `nomEtudiant.tp4.entreprise`
2. Modéliser l'organisation des employés de *PolyEntreprise*
3. Tester votre code dans une classe `Test` en créant une hiérarchie d'employés ([manager1 [manager2 [employe1, employe2]], employe3, employe4]) et en l'affichant sur la console

6 Exercice Bonus : [Set, Comparable & Cloneable]

L'entreprise *PolytechSet* a besoin d'une implémentation d'un conteneur `Java`. Voici le cahier des charges :

- le conteneur doit avoir les mêmes propriétés qu'un ensemble défini par l'interface `Set` de `java.util`
- ce conteneur doit être basé sur un `ArrayList` (donc les éléments de l'ensemble sont contenus dans un conteneur de type `ArrayList`)
- une instance de cet ensemble doit être `Cloneable`
- les instances de cet ensemble doivent être `Comparable`. Soit les ensembles a et b :
  - $a > b$ si a contient plus d'éléments que b
  - $a < b$ si a contient moins d'éléments que b
  - $a == b$ si a contient autant d'éléments que b

Questions :

1. créer un package `nomEtudiant.tp4.polytechSet`
2. Implémenter votre classe `PolytechSet`
3. Tester votre code à l'aide d'une classe `Personne`. Chaque personne contient un nom, un prénom. Deux personnes sont identiques si tous leurs attributs sont identiques.