Master Informatique **Approche Objet – 4TIN706U**
Université Bordeaux **2017 – 2018**

TD N^o2 Types de données : Arbres Binaires, Ensembles, etc.

Packages, Interfaces, Classes abstraites

Exercices

Dans ce TP, nous allons utiliser des arbres binaires pour implémenter des ensembles d'objets. Dans un ensemble d'objet, tout objet ne doit apparaître qu'une seule fois, ce qui suppose qu'on puisse comparer les éléments entre eux.

Interfaces

1. Créer un *package* `set`
2. Créer dans ce *package* les interfaces `IQueue` `IBinaryTree`, `ISet`, `IComparable`, `IIterator` et `IIteratable`.
   - `IQueue` est une file d'objets avec accès en mode FIFO ((*First In First Out*).
     Les méthodes de `IQueue` sont :
     - `void push_front(Object o);`
     - `Object pop_back();`
     - `boolean empty();`
     - `void clear();`
     - `int size();`
   - `IBinaryTree` est un arbre binaire d'éléments de type `Object`.
     Les méthodes de `IBinaryTree` sont :
     - `Object getData()`
     - `void setData(Object data)`
     - `IBinaryTree getLeftChild()`
     - `IBinaryTree getRightChild()`
     - `void setLeftChild(IBinaryTree leftChild)`
     - `void setRightChild(IBinaryTree rightChild)`
     - `boolean isLeaf()`
     - `int getHeight()`
     - `int getNumberNodes()`
   - `ISet` est un ensemble d'éléments de type `Object`.
     Les méthodes de `ISet` sont :
     - `boolean contains(Object element)`
     - `ISet union(ISet set)`
     - `ISet intersect(ISet set)`
     - `ISet diff(ISet set)`
     - `void add(IComparable element)`
     - `void remove(IComparable element)`
     - `int size()`
     - `boolean isEmpty()`

— `IComparable` est un objet comparable à un autre suivant un ordre qui sera donné en fonction de sa nature (nombre, caractère, chaîne de caractères, Objet complexe, etc).
La seule méthode de `IComparable` est `int compareTo(Object other)` dont l'implémentation future permettra de donner un nombre (-1 pour plus petit, 1 pour plus grand et 0 pour égal).
— `IIterator` énumère les objets d'un ensemble ou d'un arbre.
Les méthodes de `IIterator` sont :
— `boolean hasNext()`
— `Object next()`
la méthode `boolean hasNext()` renvoie `true` tant que l'énumération n'est pas terminée. la méthode `Object next()` renvoie un nouvel objet si l'énumération n'est pas terminée.
— Un objet `IIterable` est un objet qui possède un itérateur (par exemple un ensemble).
La seule méthode de `IIterable` est `IIterator iterator()` qui renvoie un itérateur.

Classe abstraite

3. Créer, toujours dans le même *package*, la classe abstraite `BinaryTree` qui implémente l'interface `IBinaryTree`.
On y implémentera les constructeurs `BinaryTree(Object data, BinaryTree leftChild, BinaryTree rightChild)` et `BinaryTree(Object data)` (pour créer une feuille simple).
On y implémentera aussi la méthode `String toString()` qui retourne un chaîne de caractère permettant l'affichage de l'arbre.

Itérateurs

4. Créer, toujours dans le même *package*, la classe `BinaryTreeQueue` qui implemente l'interface `IQueue`. Un objet `BinaryTreeQueue` composera une table de `BinaryTree`. Il sera nécessaire d'y implémenter la méthode `void allocate()` pour agrandi le tableau (l'usage est de doubler la taille de ce tableau à chaque fois qu'il est trop petit).
5. Créer, toujours dans le même *package*, la classe `BreadthFirstIterator` qui implemente l'interface `IIterator`. Un objet `BreadthFirstIterator` composera une file d'arbres binaires `BinaryTreeQueue`.
Dans le constructeur, on initialise la file avec la racine de l'arbre.
A chaque appel de *next()*, on prend le noeud suivant, x, dans la file. Avant de retourner x, on ajoute dans la file le fils gauche et le fils droit de x, s'ils ne sont pas vides.
L'énumération des nœuds de l'arbre est terminée lorsque la file est vide.
6. Créer la classe `BinarySearchTree` qui implémente un arbre binaire de recherche.
Cette classe implémentera les méthodes `boolean contains(IComparable element)` et `void add(IComparable element)`, et redéfinie les méthodes `setData`, `setLeftChild` et `setRightChild` pour provoquer un message d'erreur indiquant que cette opération n'est pas disponible.
Les valeurs sont ajoutées dans l'arbre de telle sorte que pour tout nœud x, les valeurs contenues dans le sous-arbre gauche sont inférieures à la valeur du nœud x et les valeurs contenues dans le sous-arbre droit sont supérieures à la valeur du nœud x.
7. Créer la classe `BstSet` qui implémente l'interface `ISet` et qui étend la classe `BinarySearchTree`.
8. Dans la *package* par défaut, créer la classe `Main` qui contiendra la méthode `main`.

— Y construire des arbres, des ensembles et les manipuler.
— Ajouter une classe `Employee` qui implémente l'interface `IComparable` et contient deux attributs, `Nom` et `Salaire`. Tester deux versions de la fonction de comparaison, une selon le nom, une selon le salaire. Construire un ensemble d'objets instances de cette classe. Manipuler cet ensemble.

Créer un *package* `generics`

Faire ce même travail en utilisant des classes génériques et les classes `Comparable<T>`, `Iterator<E>` et `Iterable<T>` de l'API Java. Quel apport cela procure-t-il ?

Polymorphisme

Le but de cette deuxième partie de TP est de travailler sur le polymorphisme.

Pour commencer, mettons nous dans le cadre d'une entreprise. Chaque métier de cette entreprise possède sa propre classe. À chaque métier correspond un indice qui multiplié par le salaire de base de l'entreprise nous donne le salaire associé à ce métier. Chaque instance d'une classe correspond à un employé qui est identifié par son nom. Toutes les classes héritent d'une classe abstraite `Employee` qui a pour méthodes :

— `double getWage();`
— `String getName();`
— `String toString()` qui renvoie une chaîne de caractères contenant le nom et le salaire de l'employé.

À titre d'exemples nous créons les trois classes suivantes :

— `Warehouseman` dont le salaire se calcule comme la plupart des autres métiers : indice * salaire de base ;
— `SalesAssistant` dont le salaire inclus également 20% de son chiffre d'affaires qui est un attribut de la classe ;
— `Guard` dont le salaire peut inclure une prime de risque que l'on ajoute à la volé dans le calcul du salaire.

Pour quelle classe avons nous une surcharge et pour quelle autre avons nous un polymorphisme ?

L'héritage et le polymorphisme possèdent quelques subtilités qu'il est important de comprendre. Considérons les deux classes `A` et `B` suivantes

```
1   class A
    {
3       int f(A a)
        {
5           return 1;
        }
7   }

9   class B extends A
    {
11      int f(A a)
        {
13          return 2;
        }
15
        int f(B b)
17      {
            return 3;
```

```
19          }
        }
```

Calculez maintenant le retour de chacun des appels suivant

```
        public static void main(String[] astrArgs)
2       {
            A a = new A();
4           A ab = new B();
            B b = new B();
6
            // Partie a
8           System.out.println( a.f(a) );
            System.out.println( a.f(ab) );
10          System.out.println( a.f(b) );

12          // Partie ab
            System.out.println( ab.f(a) );
14          System.out.println( ab.f(ab) );
            System.out.println( ab.f(b) );
16
            // Partie b
18          System.out.println( b.f(a) );
            System.out.println( b.f(ab) );
20          System.out.println( b.f(b) );
        }
```

Si vous n'avez pas obtenu les résultats attendus, faites un tour sur la page `http://mess.genezys.net/polymorphisme`.