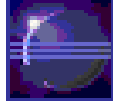
	TP Collections en JAVA	
---	---	---

Objectifs	Temps alloué	Outils
<ul style="list-style-type: none"> - Apprendre à manipuler les collections génériques 	3h00	Eclipse

Exercice 1 : Pile et collection générique (ArrayList)

Vous disposez maintenant du code suivant représentant l'implémentation contigu d'une pile.

```
public class PileTab {
    int pos;
    int [] tab;

    public PileTab(){
        tab = new int[4];
        this.pos = 0;
    }

    public void add(int e){
        if (this.pos == tab.length){
            System.out.println("Ajout impossible de "+e);
            //System.exit(-1);
        }
        else {
            tab[this.pos] = e;
            this.pos ++;
        }
    }

    public int remove(){
        int elem = tab[this.pos-1];
        this.pos --;
        return elem;
    }

    public boolean estVide(){
        return this.pos == 0;
    }

    public int size(){
        return this.pos;
    }
}
```

```

public void affiche() {
    for (int i = 0; i < this.pos; i++) {
        System.out.println(this.tab[i]);
    }
}

public int getSommet() {
    return this.tab[pos-1];
}
}

```

Question :

- 1) Testez cette classe dans votre programme principal
- 2) Transformer ce code en un type paramétré PileGen de façon à ce qu'on puisse ajouter n'importe quel type d'objet dans la pile. Le champ tableau devient un champ ArrayList <T> dont la taille est extensible, donc vous n'avez plus besoin du champ pos, vous pouvez utiliser directement la méthode size() de l'ArrayList.
- 3) Dans la méthode d'affichage, vous utiliserez les 3 méthodes de parcours vu en cours (boucle for, for each et itérateur).
- 4) Testez La classe PileGen dans votre programme principal en

Exercice 2 : Pile et collection générique (LinkedList)

Voici une interface définissant un type abstrait "Pile de <A>" avec les fonctionnalités classiques d'une pile :

```

public interface IPile <A> {
    boolean estVide();
    void empile(A a);
    A depile(); // retourne l'élément en sommet de pile et dépile
    int nbElements();
    A sommet(); // retourne le sommet de pile mais ne le dépile pas
}

```

- 1) Écrivez une classe générique **CPile** qui implémente l'interface **IPile**. Vous stockerez les éléments de la pile dans une liste chaînée (instance de **java.util.LinkedList**, voir en annexe quelques méthodes publiques de cette classe).
- 2) Écrivez un petit programme qui crée et manipule des piles en instanciant la classe générique de différentes façons (par exemple pile de String, pile de Integer, ...).

Exercice 3 : Compréhension de l'architecture des tâches

L'architecture des tâches est donnée à la figure 1 où le détail des classes TacheElementaire et TacheComplexe n'est pas donné.

Une tâche est caractérisée par un nom et un coût. Une tâche est soit une tâche élémentaire, soit une tâche complexe qui est alors composée de sous-tâches.

Il est ainsi possible d'ajouter une sous-tâche à une tâche complexe, ajouter(Tache) ou de supprimer une sous-tâche, supprimer(Tache). Le coût d'une tâche complexe est la somme des coûts des tâches qui la composent.

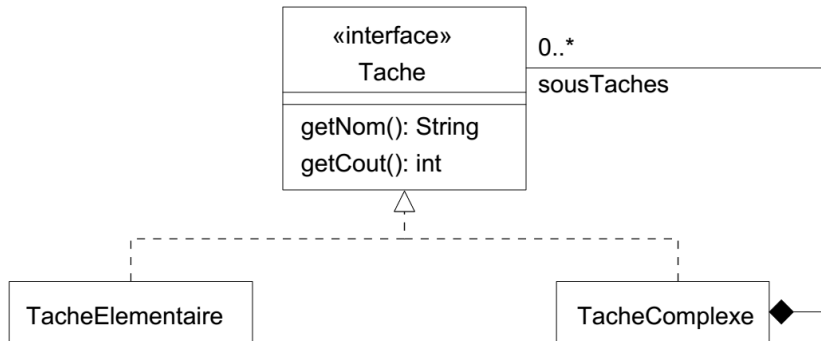


FIG. 1 – Architecture des tâches hiérarchiques

- 1) Écrire en Java la classe TacheElementaire qui est une réalisation de l'interface Tache.

```

public interface Tache {
    /** Obtenir le nom de la tâche. */
    String getNom();

    /** Obtenir le coût de la tâche. */
    int getCout();
}
  
```

Définition d'une tâche complexe

Nous nous intéressons maintenant à la classe TacheComplexe, en particulier à sa relation avec l'interface Tache. Une tâche complexe est composée d'un nombre quelconque de tâches. On décide d'utiliser l'interface `java.util.Collection` pour stocker les sous-tâches. On l'utilisera bien entendu dans sa version générique.

Comme on souhaite pouvoir parcourir toutes les sous-tâches d'une tâche complexe, la classe `TacheComplexe` réalise l'interface `java.lang.Iterable`.

- 2) Indiquer quel est le principal intérêt de la généricité.
- 3) Écrire en Java la classe `TacheComplexe`.
- 4) Écrire un programme qui crée des tâches et des tâches complexes et qui affiche leur coût total.

Exercice 4 : Collection générique `HashMap`

L'objectif de cet exercice est de réaliser un annuaire pour mémoriser des numéros de téléphone et des adresses. Chaque entrée est représentée par une fiche à plusieurs champs : un nom, un numéro et une adresse.

Travail à faire :

- 1) Écrire la classe `Fiche`. Celle-ci devra avoir un constructeur à 3 paramètres pour initialiser l'ensemble des attributs de la classe et 1 constructeur à 1 paramètre auquel on passera le nom (*Les autres champs sont initialisés par défaut à -1 (numéro) et null (adresse)*).

La classe **Annuaire** comporte une table associative (`HashMap<String , Fiche>`) qui sera faite d'associations (un nom , une `Fiche`).

- 2) Écrire la classe `Annuaire` dont le constructeur permettra la création d'une table associative de type `HashMap`. Celle-ci doit comporter les méthodes suivantes :
 - a. `public void getNbcontacts()` qui retourne le nombre de contacts dans l'annuaire
 - b. `public void addContact(Fiche f)` qui ajoute un contact à partir d'une fiche passé en paramètre.

- c. `public void addContact(String s, int n, String a)` qui ajoute un contact à partir d'un nom, d'un numéro et d'une adresse passés en paramètre.
- d. `public void addContact(String s)` qui ajoute un contact à partir du nom passé en parameter (numéro et adresse par défaut)
- e. `public int getnumero(String name)` qui retourne le numéro de telephone associé au nom passé en paramètre
- f. `public void affiche()` qui affiche les contacts dans l'annuaire.

3) Réalisez une classe Test afin d'y tester toutes vos méthodes

Annexe

la classe LinkedList

java.util

Class LinkedList<E>

java.lang.Object

java.util.AbstractCollection<E>

java.util.AbstractList<E>

java.util.AbstractSequentialList<E>

java.util.LinkedList<E>

Type Parameters :

E - the type of elements held in this collection

Quelques méthodes publiques :

- **LinkedList()** Constructs an empty list.
- **void addFirst(E o)** Inserts the given element at the beginning of this list.
- **E element()** Retrieves, but does not remove, the head (first element) of this list.
Throws : `NoSuchElementException` - if this queue is empty.
- **E getFirst()** Returns the first element in this list.
Throws: `NoSuchElementException` - if this list is empty.