# Fiche d’exercices – Réutilisation

Pour chaque exercice ci-dessous, fournir le code **Java**, ainsi que la **modélisation UML** (projet *NetBeans* & *Visual Paradigm for NetBeans* compacté sur *Nom-FicheReuse-ExerciceX.zip*).

1. Définir une classe Personne permettant la comparaison entre deux objets de cette classe. La comparaison doit se faire en fonction de l’ordre alphabétique selon le nom de famille.
2. Analyser et corriger le code ci-dessous. Justifier vos positions.

|  |  |
| --- | --- |
| 12345678910111213141516171819202122232425 | **public class Panier {**  private ArrayList produits = new ArrayList(); private String[] livraison = null; **public void ajouterProduit**(String codeProd, String nomProd, int qte, float prixUnit) { this.produits.add(new Produit(codeProd, nomProd, prixUnit\*qte)); }  **public float calculerPrix() {** float prix = 0; for (int i=0; i<produits.size(); i++) { Produit p = (Produit) produits.get(i); prix += p.getPrix(); } return prix; } **public void setLivraison(String[] adresse)** { this.livraison = adresse; } **public String[] getLivraison()** { return this.livraison; } **public void imprimer()** { System.out.println("Facture "); for (int i=0; i<produits.size(); i++) { Produit p = (Produit) produits.get(i); System.out.println(p); } System.out.println("Prix total : "+ this.calculerPrix()); }} |
| 123456789101112 | **public class Produit {** private String code; private String nom; private float prix; **public Produit(String c, String n, float p)** { code = c; nom = n; prix = p; } **public float getPrix()** { return this.prix; } **public String toString ()** { return code + " " + nom + " " + prix + "€"; }} |

1. Etendre la classe Etudiant (proposée dans la fiche d’exercice précédente) de manière à ce qu’on puisse enregistrer un nombre quelconque de notes (le nombre doit être renseigné par l’utilisateur).

Ecrire une application permettant à un enseignant de saisir les notes d’un groupe d’étudiants. L’application doit, par la suite, effectuer la restitution des informations (moyenne, note maximale, note minimale) pour chaque étudiant et pour tout le groupe.

A partir de la classe proposée, répondre aux questions suivantes :

1. Avez-vous pu réutiliser la classe précédemment définie sans la modifier ?
2. Quelles ont été les difficultés majeures pour la réutilisation ?
3. Etendre la classe TableauEntier (proposée lors de la fiche d’exercice précédente) de manière à ce que l’utilisateur puisse remplir le tableau séquentiellement et non plus de manière « aléatoire ». Désormais, l’utilisateur n’a plus besoin d’indiquer la position qu’il souhaite remplir, car les cases doivent être remplies les unes après les autres. Par contre, l’utilisateur doit pouvoir supprimer n’importe quelle case du tableau, juste en indiquant sa position. Utiliser les mécanismes de surcharge et de redéfinition si nécessaire.

1. Etendre la classe TableauObjets proposée précédemment de manière à ce qu’on puisse créer un nouveau tableau à partir d’un tableau existant, et qu’on puisse comparer deux tableaux (vérifier si deux tableaux sont égaux ou pas de tout).

A partir de la classe proposée, répondre aux questions suivantes :

1. Comment avez-vous fait pour cloner un tableau ?
2. Lors de la comparaison entre deux tableaux, la classe proposée compare-t-elle le contenu du tableau ? Comment le fait-elle ?
3. Définir et implémenter deux interfaces représentant, respectivement, les structures de données « Pile » (Stack) et « File d’attente » (Queue).
4. A l’aide de l’interface définie précédemment, proposer une classe qui implémente la notion de « File d’attente ». Cette classe doit également implémenter l’interface « Cloneable » proposée par Java.
5. A partir de l’interface définie dans l’exercice précédent, proposer différentes implémentations pour la notion de pile, dont une utilisant un tableau. Définir alors une application qui permet à l’utilisateur de choisir une implémentation parmi celles proposées et de l’utiliser ensuite pour empiler des objets (String, par exemple) et pour les dépiler.

A partir des classes proposées, répondre aux questions suivantes :

1. Comment avez-vous fait pour gérer les différentes implémentations dans l’application ?
2. Dans l’implémentation avec un tableau, comment avez-vous géré la taille du tableau (inconnue au départ) ? Et son redimensionnement en cas de pile pleine ?
3. Modifier une des implémentations proposées dans les exercices précédents (au choix) afin qu’elle puisse s’intégrer au *framework* Collection, et notamment, à l’interface « Queue ».
4. Comment avez-vous procédé ? Quel a été l’impact sur le code ?
5. Pourriez-vous insérer l’interface « Pile » elle-même dans le *framework* Collection ? Comment le feriez-vous ? Quelles seraient les conséquences sur le code existant ?
6. Définir une interface représentant la notion de Liste doublement chaînée.
	1. Serait-il possible d’intégrer l’interface proposée au *framework* Collection proposé par l’API Java ? Comment se ferait-elle cette intégration ? Quels en seraient les conséquences ?
7. Définir une classe implémentant l’interface Liste doublement chaînée. Permettre à cette classe d’être parcourue à l’aide d’un « Iterator ».
8. Construire une application capable de saisir un certain nombre de lignes renseignées par l’utilisateur et de les enregistrer dans une liste doublement chaînée (celle proposée dans l’exercice précédent). L’application doit permettre à l’utilisateur de parcourir cette liste (vers l’avant et vers l’arrière).
9. Etendre l’application de manière à enregistrer la liste renseignée par l’utilisateur dans un fichier texte.
10. Etendre l’application précédente pour qu’elle puisse aussi lire la liste à partir d’un fichier texte.
11. Construire, à partir de l’application précédente, une application qui construit un index de mots : pour chaque mot, on doit afficher le numéro des lignes dans lesquelles le mot apparaît.