Module 1 Développer l'interface d'une application informatique

Programmation Objet P2	Séance	S04	Activité	A-001

A l'issue de cette activité, vous devrez être capable d'identifier les mécanismes de gestion d'ensembles d'objet. Vous aurez notamment mis en œuvre :

- La gestion de liste d'objets au sein de listes et collections
- L'identification d'un type et sa conversion
- L'ajout, l'extraction et la suppression d'objets au sein d'une collection

Sommaire de l'activité proposée :

1 Un type spécialisé pour gérer un ensemble homogène d'objets

Dans cet exercice consacré aux collections, nous allons faire évoluer notre bibliothèque consacrée aux salariés pour prendre en charge la gestion d'un ensemble de salariés.

Nous allons ici mettre en œuvre conjointement la notion de collection générique et celle de l'héritage pour proposer un nouveau type « Salarie**s** ».

Créez un nouveau code source Salaries.cs et la définition du type Salaries en le faisant hériter du type collection **List<T>** où T est de type Salarie.

En héritant du type List<T>, vous héritez d'un certain nombre de comportements déjà implémentés dans cette classe de base. Vous pourrez, si nécessaire, leur substituer votre propre implémentation.

Vous devez programmer les 3 fonctionnalités suivantes : Ajout d'un salarié sans doublon, extraction d'un salarié par son matricule, la suppression d'un salarié.

1. Ajouter un salarié.

   Cette méthode prend en argument une instance de type Salarie.

   Il existe déjà une méthode Add héritée de la classe List<T> de base. Nous devons modifier celle-ci afin de nous assurer, lors de l'ajout d'un salarié, qu'il n'est pas déjà présent.

   La méthode de la classe mère n'étant pas définie comme substituable (virtual), nous devons définir une nouvelle méthode qui viendra remplacer celle-ci.
   Nous allons la définir en ayant recours au mot clé new qui précise le remplacement explicite de la méthode de la classe mère.

   Pour vérifier si un salarié n'est pas déjà présent, il vous faut comparer chaque instance de salarié présente dans la liste et la comparer à celle que vous souhaitez ajouter.

   L'occasion ici de vérifier si la surcharge de la méthode Equals a été bien implémentée.

   Extrait pour exemple où une méthode Add est redéfinie pour permettre l'implémentation d'un comportement particulier :

```
public class Type
 {
     public int MaPropriete { get; set; }
 }

public class Types : List<Type>
{
    public new void Add(Type instanceType)
    {

    }
}
```

   Pour itérer sur la liste des objets présents ayez recours au cycle **foreach**.

   Si le salarié est absent de la liste, il convient de l'ajouter en invoquant la méthode de base **base.Add**(instanceSalarie).

2. Extraire un salarié
   Vous devez fournir une méthode qui retourne un objet Salarié à partir de son matricule.

3. Supprimer un salarié dans la liste
   Nous devons avoir 2 implémentations.
   Une première qui prend en paramètre une instance de salarié.
   Une seconde qui reçoit en argument le matricule du salarié.
   Il existe déjà une première version dans la classe de base. Est-il nécessaire de modifier son comportement ?

1.1 Version 2

Il n'est pas certain que le recours au conteneur de type List<T> soit le plus pertinent.

Si nous souhaitons obtenir des ensembles sans doublon, nous pouvons substituer à notre composant List<T> un composant HashSet. La méthode Add prendra donc en compte cet objectif de ne pas introduire de doublon dans la liste.

Pour cela elle s'appuiera sur le code de hachage généré par la méthode GetHashCode.

Tester cette nouvelle implémentation en développement une deuxième version de votre type dérivé de HashSet.

En conclusion, vous devez choisir le type de conteneur avec soin.