PROGRAMMATION ORIENTÉE OBJET

Informatique de gestion - bloc 2

Haute-École de Namur-Liège-Luxembourg

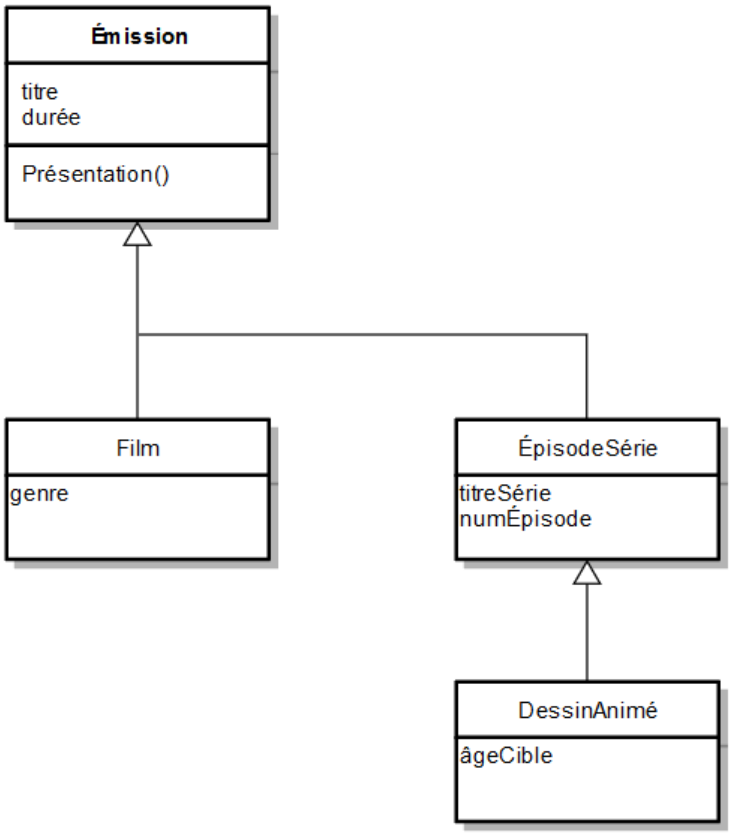
Laboratoire 3 : héritage

Objectifs

- Se familiariser avec la gestion de l'héritage en C# (mots-clefs override, virtual, new)
- Utiliser des interfaces et des classes abstraites en C#
- Mettre en œuvre l'héritage avec des propriétés
- Appliquer toutes ces connaissances dans le cadre d'exercices pratiques

Exercice 1 : Programme TV et émissions

Le but de ce laboratoire est d'aborder l'utilisation de l'héritage en C#, sur base d'un exemple traitant d'émissions TV comme vous pouvez en lire dans Télépro. Cet exemple s'articule principalement autour de deux classes : la classe Émission qui servira de classe-mère à toute une hiérarchie de classes et la classe utilitaire `UtilTV` qui regroupera une série de méthodes statiques permettant d'exploiter les émissions.



Étape 1 : les premières classes

Définissez une classe `Émission` qui reprend les attributs suivants. Déclarez les attributs en utilisant la visibilité `protected` pour qu'ils puissent être utilisés dans les classes-filles.

- titre : le titre de l'émission
- durée : la durée (en minutes) de l'émission

Ajoutez un constructeur standard ainsi qu'une méthode `Présentation` qui renvoie une chaîne de caractères constituée du titre de l'émission suivi de sa durée en minutes entre parenthèses.

À l'intérieur de `UtilTV`, définissez une méthode statique `AfficheProgramme` recevant un tableau d'émissions ou tout simplement plusieurs arguments de type émission et affichant leurs présentations (via la méthode `Présentation`) dans l'ordre.

Dans la méthode Main de la classe Program, définissez deux ou trois émissions qui pourront servir pour les tests.

Étape 2 : premier héritage

Parmi les émissions, on distingue les films. Créez une classe `Film` qui hérite de `Émission`. Pour ce faire, utilisez l'en-tête suivante.

```
public class Film : Émission
```

En plus d'un titre et d'une durée, le film aura un genre (western, aventures, fantastique…). Dans la classe `Film`, déclarez ce nouvel attribut.

Pour définir un constructeur standard pour `Film`, tout comme en Java, il faut réaliser un appel au constructeur de la classe-mère. En Java, cet appel, s'il est explicite, doit être placé au début du code du constructeur. En C#, comme pour les appels entre constructeurs d'une même classe, cet appel s'ajoute dans l'en-tête du constructeur.

Définissez un constructeur standard pour Film en utilisant l'en-tête suivante.

```
public Film (string titre, int durée, string genre) : base(titre, durée)
```

Notez l'utilisation du mot-clef « base » pour se référer à la classe mère.

Ajoutez une méthode `Présentation` à la classe Film. Celle-ci renverra une description similaire à celle d'une émission mais suivie d'une virgule, d'une espace et du genre du film en question. Dans le code de cette méthode, utilisez `base.Présentation()` pour obtenir la valeur renvoyée par la méthode de la classe-mère.

Dans la méthode Main, créez un ou deux films puis ajoutez un appel à `UtilTV.AfficheProgramme` en mélangeant émissions et films dans ses arguments. Le résultat correspond-il à ce que vous attendiez ?

Dans votre programme, notez que Visual Studio a souligné le mot « Présentation » en vert dans la classe Film. Positionnez votre curseur sur le soulignement et lisez le message d'erreur.

```
« Film.Présentation() hides inherited member Émission.Présentation() ;
use the new keyword if hiding was intended. »
```

Utilisez Visual Studio pour appliquer la correction proposée ou réalisez-la manuellement, de manière à obtenir l'en-tête suivante.

```
public new string Présentation ()
```

Effectuez les tests à nouveau. Le résultat reste le même… Grâce au message d'erreur copié ci-dessus, comprenez-vous ce qui se passe ? Dans tous les cas, poursuivez avec l'étape suivante (où la réponse est présentée).

Étape 3 : héritage et redéfinition

Au point de vue de l'héritage des méthodes, C# et Java prennent des points de vue diamétralement opposés.

En Java, si la classe-fille contient une méthode déjà définie dans la classe-mère, il s'agit par défaut d'une redéfinition. Concrètement, cela signifie que, grâce au principe de la liaison dynamique (dynamic binding), un appel à cette méthode exécutera le code correspondant à la classe de l'objet lors de l'exécution.

Ainsi, en Java, l'appel `émission.Présentation()` dans `UtilTV.AfficheProgramme` exécuterait la méthode définie dans Émission pour les émissions et celle définie dans Film pour les films.

En C#, l'option par défaut est l'inverse : si on ne précise pas vouloir une redéfinition, C# suppose que la nouvelle méthode (celle de la classe-fille) ne peut pas se substituer à celle de la classe-mère.

Ainsi, dans l'exemple ci-dessus, l'appel `émission.Présentation()` exécute toujours la méthode d'`Émission`, jamais celle de `Film` !

Si on désire l'autre comportement en C#, il faut le préciser **explicitement**, ce qui se fait en deux temps.

- Tout d'abord, il faut déclarer la méthode du parent « virtuelle », en utilisant le mot-clef `virtual`.
- Ensuite, il faut préciser que la méthode de la classe fille redéfinit (en anglais : `override`) la méthode de l'ancêtre.

Modifiez votre code pour que les en-têtes des méthodes Présentation soient les suivantes.

```
public virtual string Présentation()          // dans Émission
public override string Présentation()         // dans Film
```

Une fois tous ces changements effectués, refaites le test précédent et observez le résultat.

Étape 4 : séries

Créez une nouvelle classe `ÉpisodeSérie` pour les séries TV, qui sont considérées comme des cas particuliers d'émissions. Chaque objet représentera un épisode d'une série. Dans ce cas-ci, en plus du titre de l'émission (c'est-à-dire le titre de l'épisode), on retiendra également le titre de la série (qui peut être différent du titre de l'épisode) et le numéro de l'épisode.

Créez un constructeur standard et définissez une méthode Présentation qui présente la série au format suivant.

<*Titre de la série*> : <*Titre de l'épisode*> (*durée* minutes)

Étape 5 : dessin animé

Ajoutez une 4e classe `DessinAnimé` qui hérite de `ÉpisodeSérie`. On considère, ici, les séries dessins animés, comme par exemple « The Simpsons » ou « Goldorak ». Pour un dessin animé, on retient aussi l'âge-cible (un entier).

Créez un constructeur standard.

Ajoutez dans `UtilTV` la définition suivante.

```
public static void PrésenteDA (params DessinAnimé[] das)
{
  foreach (DessinAnimé da in das)
    Console.WriteLine(da.Présentation());
}
```

Pourrez-vous compléter le code de la classe `DessinAnimé` pour que le test

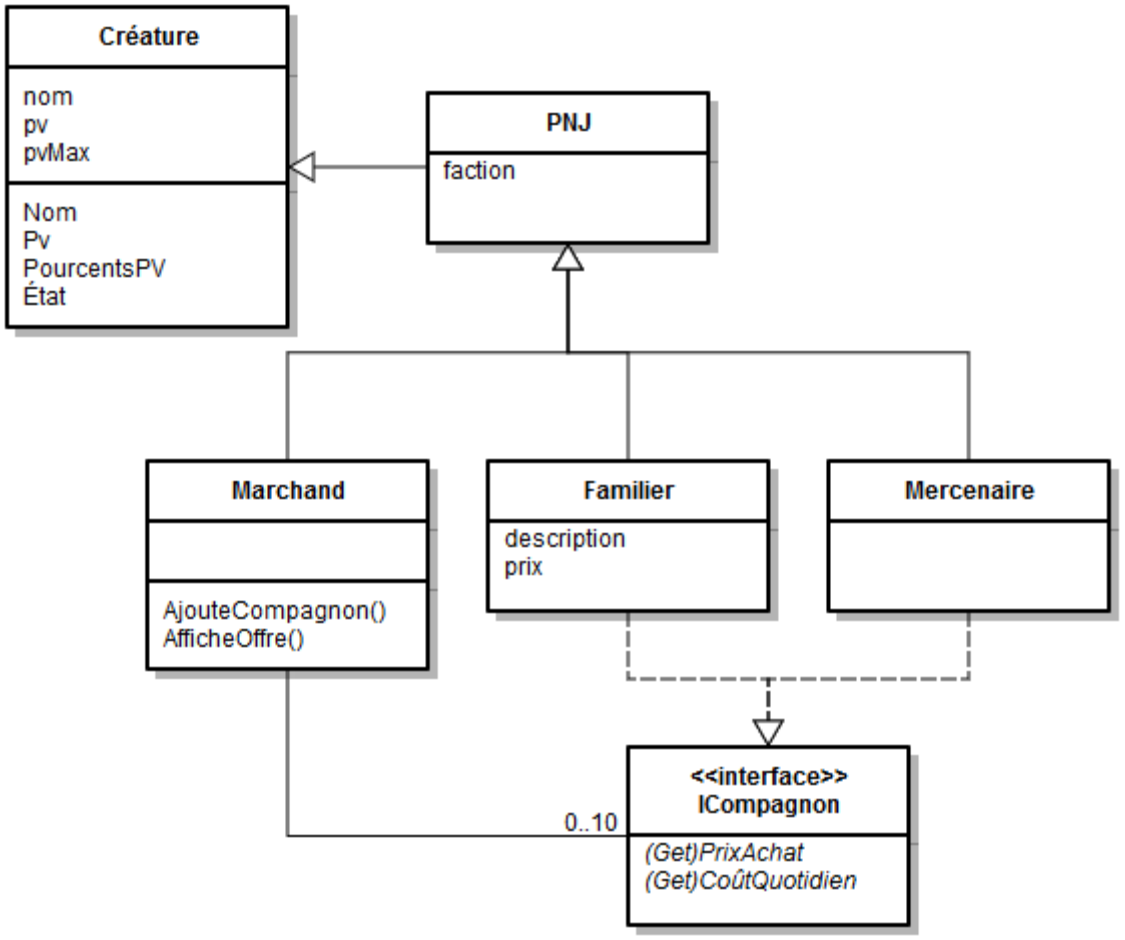
```
Émission e1 = new Émission("Motus", 50);
Émission e2 = new Émission("Des chiffres et des lettres", 45);
Film f1 = new Film("Bilbo the Hobbit", 182, "fantastique");
Film f2 = new Film("Le bon, la brute et le truand", 178, "western");
Série s1 = new ÉpisodeSérie("Monsters", 47, "The Walking Dead", 801);
Série s2 = new ÉpisodeSérie("The Interrogation", 52, "Designated
Survivor", 6);
DessinAnimé da1 = new DessinAnimé("Simprovised", 24, "Simpsons", 2721,
16);
DessinAnimé da2 = new DessinAnimé("Droids in Distress", 22, "Star Wars
Rebels", 3, 8);

UtilTV.AfficheProgramme(e1, e2, e2, f1, f2, s1, s2, da1, da2);
Console.WriteLine("....");
UtilTV.PrésenteDA(da1, da2);
```

affiche exactement les lignes suivantes ?

```
Motus (50 minutes)
Des chiffres et des lettres (45 minutes)
Des chiffres et des lettres (45 minutes)
Bilbo the Hobbit (182 minutes), fantastique
Le bon, la brute et le truand (178 minutes), western
The Walking Dead : Monsters (47 minutes)
Designated Survivor : The Interrogation (52 minutes)
Simpsons : Simprovised (24 minutes)
Star Wars Rebels : Droids in Distress (22 minutes)
....
Enfants de 16 ans : Simprovised (Simpsons numéro 2721)
Enfants de 8 ans : Droids in Distress (Star Wars Rebels numéro 3)
```

Exercice 2 : Personnages d'un jeu

L'objectif de cet exercice est d'utiliser l'héritage pour implémenter quelques classes représentant des créatures dans un jeu, dont certaines vont pouvoir être achetées ou embauchées par les joueurs.



Étape 1 : la classe Créature

La classe `Créature` contiendra des attributs privés correspondant au nom d'une créature (`nom`), à son nombre actuel de points de vie (un entier `pv`) et à son nombre maximum de points de vie lorsqu'elle est en pleine santé (un entier `pvMax`).

Ajoutez-lui une propriété publique `Nom` (en lecture et en écriture) permettant d'obtenir le nom de la créature, ou la chaîne "<aucun nom>" si on ne lui a donné aucun nom (`null`) ou juste un nom vide (chaîne de caractères vide).

Ajoutez également une propriété publique `Pv` (en lecture et en écriture) permettant d'obtenir le nombre de points de vie actuel et de le modifier. Après la modification, le nombre de points de vie devra se trouver (au sens large) entre 0 et le nombre maximal de points de vie. Si on tente d'utiliser une valeur située en-dehors de ces bornes, on utilisera la borne la plus proche (0 en cas de valeur négative ; `pvMax` en cas de valeur dépassant le nombre de points de vie maximal).

Ajoutez une propriété publique `PourcentsPV` (en lecture seule) renvoyant le pourcentage de points de vie que la créature possède (par rapport à son nombre de points de vie maximal).

Ajoutez un constructeur standard recevant les trois arguments (si le nombre maximal de points de vie est inférieur ou égal à 1, on considérera qu'il vaut 1 ; le nombre de points de vie sera ajusté à une valeur acceptable, en suivant les mêmes règles que ci-dessus, ce qui se fera automatiquement car vous utiliserez la propriété).

Ajoutez un second constructeur ne recevant que le nom et le nombre maximal de points de vie (initialement, on supposera alors que la créature est en pleine santé).

Complétez la classe `Créature` en lui ajoutant une propriété `État` en lecture seule renvoyant une présentation de la créature sous la forme d'une chaîne de caractères au format suivant (indiquant le nom, le nombre actuel de points de vie, le nombre maximum de points de vie et le pourcentage que cela représente, le tout suivi de la chaîne « MORT » si les points de vie sont à zéro) :

```
Hubert (15/15 : 100.00%)
Hubert (10/15 : 66.67%)
Abu (0/1 : 0.00%) - MORT
```

Notez que, si `val` est un réel, vous pouvez en obtenir une représentation avec 2 décimales sous la forme d'un caractère en écrivant `val.ToString("F2")`.

Pour obtenir le nom, le nombre de points de vie actuel et le pourcentage que cela représente, assurez-vous de bien faire appel aux propriétés définies plus haut !

Étape 2 : la classe PNJ

Parmi les créatures, on trouve des PNJ (personnages non joueurs, incarnés par l'ordinateur). Créez une classe `PNJ` qui hérite de `Créature`.

Chaque PNJ sera caractérisé par une faction (une simple chaîne de caractères). Pour en faire automatiquement une propriété publique avec un getter et un setter sans vérification spécifique et laisser le compilateur créer et gérer lui-même l'attribut privé caché derrière celle-ci, utilisez la syntaxe suivante.

```
public string Faction {get; set;}
```

Dans ce cas, en C#, le programmeur n'est même plus obligé de déclarer l'attribut privé. En encodant la ligne ci-dessus, automatiquement, l'environnement sait créer un attribut faction de type private. Vous pouvez donc effacer la déclarative `private string faction ;`.

Créez deux constructeurs : un précisant les points de vie actuels en plus du nom et de la faction et l'autre précisant juste le nom et la faction.

Étape 3 : la classe Familier

Parmi les PNJ, on trouve des familiers, des petites créatures purement décoratives qui peuvent accompagner les personnages. Créez une classe `Familier` qui hérite de la classe `PNJ` et qui possède deux propriétés : `Description` (chaîne de caractères) et `Prix` (un entier indiquant le prix d'achat).

Pour `Description`, vous pouvez utiliser une propriété auto-implémentée (sans condition supplémentaire dans le getter et le setter donc) ; pour `Prix`, assurez-vous qu'on n'y place jamais une valeur négative (si on tente de le faire, le prix ne sera pas modifié).

Ajoutez à cette classe un unique constructeur où on renseignera le nom du familier, sa description et son prix. Tous les familiers possèdent exactement 1 point de vie et appartiennent à la faction « alliés ».

À ce moment-ci de votre implémentation, le test

```
PNJ hubert = new PNJ("Hubert", 15, "commerçants");
Familier abu = new Familier("Abu", "un petit singe", 20);

Console.WriteLine(hubert.État);
hubert.Pv -= 5;
Console.WriteLine(hubert.État);
hubert.Pv += 10;
Console.WriteLine(hubert.État);
Console.WriteLine(abu.État);
abu.Pv -= 2;
Console.WriteLine(abu.État);
```

devrait produire l'affichage suivant :

```
Hubert (15/15 : 100.00%)
Hubert (10/15 : 66.67%)
Hubert (15/15 : 100.00%)
Abu (1/1 : 100.00%)
Abu (0/1 : 0.00%)- MORT
```

Étape 4 : héritage et redéfinition de propriétés

On décide que les familiers sont désormais invincibles. Cela signifie que leur nombre de points de vie actuel sera toujours 1 et qu'il ne peut être modifié.

Dans la classe `Familier`, redéfinissez la propriété `Pv` de sorte que celle-ci indique toujours 1 et que toute tentative de la modifier n'entraîne aucun changement.

Pour que cette nouvelle définition soit prise en compte lorsqu'on demande l'`État` d'un familier, vous devrez modifiez votre code en ajoutant deux mots clefs (les propriétés fonctionnent comme des méthodes et utilisent les mêmes mots-clefs).

Une fois cette modification effectuée, le test présenté à la fin de l'étape précédente devrait produire un résultat légèrement différent : les deux dernières lignes devraient indiquer qu'Abu a tous ses points de vie !

Étape 5 : une interface pour les compagnons

Certaines créatures peuvent accompagner les héros. On les appelle collectivement des compagnons. Les familiers (introduits ci-dessus) sont une catégorie particulière de compagnons.

Lorsqu'un héros acquiert un compagnon, il doit généralement payer un prix initial (le coût d'embauche) puis, dans certains cas, dépenser chaque jour une somme supplémentaire pour continuer à employer le compagnon.

Définissez une interface appelée `ICompagnon` et demandant l'existence des deux méthodes suivantes : `GetPrixAchat()` donnant le prix d'achat initial d'un compagnon (un entier) et `GetCoûtQuotidien()` donnant le coût supplémentaire à payer quotidiennement.

Syntaxiquement, la définition d'une interface se réalise en C# de la même manière qu'en Java. Entre autres, dans les deux cas, le modificateur `public` s'applique automatiquement à toutes les signatures de méthodes données. Une différence à noter toutefois : en Java, vous pouvez préciser `public` devant chacune des méthodes de l'interface (même si c'est inutile) ; en C#, vous ne pouvez pas écrire d'indicateur de visibilité dans une interface (cela déclenche une erreur).

Modifiez la classe `Familier` pour qu'elle implémente l'interface `ICompagnon`. Alors que Java distingue les mots-clefs `extends` (pour l'héritage) et `implements` (pour l'implémentation d'une interface), C# utilise la même syntaxe dans les deux cas. L'en-tête de la classe deviendra donc la suivante.

```
class Familier : PNJ, ICompagnon
```

Tout comme en Java, une classe ne peut hériter que d'au plus une classe mais elle peut implémenter autant d'interfaces que nécessaire. Vous souvenez-vous de la raison de cette restriction sur l'héritage multiple ? Si ce n'est pas le cas, revoyez votre cours de PPOO ou cherchez la réponse sur le web !

N'oubliez pas d'implémenter, dans `Familier`, les méthodes imposées par l'interface `ICompagnon` en notant qu'un familier doit être acheté initialement mais que, par la suite, les coûts d'entretien sont négligeables.

Testez le code en créant quelques familiers (un chiot, un cheval, un dragonet, au choix).

Étape 6 : les propriétés, des méthodes comme les autres !

Dans la première version de l'interface `ICompagnon`, on a utilisé des méthodes de type « getter » reposant sur les conventions utilisées en Java. Comme mentionné dans les laboratoires précédents, en C#, on travaille plutôt avec le format « propriétés ».

Modifiez l'interface `ICompagnon` pour y déclarer que toute classe implémentant `ICompagnon` devra posséder deux propriétés en getter : `PrixAchat` et `CoûtQuotidien`. Pour cela, utilisez la syntaxe suivante pour chacune de ces deux propriétés.

```
int PrixAchat { get; }
```

Modifiez ensuite le code de la classe `Familier` pour coller à cette nouvelle version de l'interface : cette fois-ci, dans la classe Familier, vous devrez définir des getters pour les propriétés `PrixAchat` et `CoûtQuotidien` afin de « respecter le contrat de l'interface ICompagnon ».

Modifiez en conséquence votre code de test et vérifiez que tout fonctionne toujours.

Étape 7 : des marchands

Parmi les PNJ, on trouve des marchands mettant à disposition des joueurs divers compagnons (des familiers mais aussi des mercenaires – voir plus loin). Définissez une classe `Marchand` qui hérite de `PNJ`.

Tous les marchands appartiennent à la faction « commerçants » et possèdent une liste de compagnons qu'ils sont prêts à « vendre » (ou dont ils sont prêts à louer les services). Cette liste sera encodée sous la forme d'un tableau d'objets implémentant l'interface `ICompagnon` (des familiers par exemple, mais pas uniquement) et comprenant au plus 10 compagnons.

Pour simplifier les choses, on supposera que tous les marchands possèdent exactement 10 points de vie (mais, contrairement aux familiers, ils ne sont pas invincibles). Ajoutez donc un constructeur permettant de créer un marchand en précisant juste son nom (initialement, le tableau de compagnons sera vide).

Ajoutez également une méthode `AjouterCompagnon` qui ajoute un compagnon au tableau des compagnons vendus (on pourra supposer qu'il n'y a pas d'erreur de dépassement).

Dans l'interface `ICompagnon`, ajoutez que chaque classe implémentant celle-ci devra posséder une propriété `Nom` renvoyant une chaîne de caractères en lecture seule. Notez qu'il n'y a rien à ajouter dans la classe car cette dernière hérite déjà de la propriété Nom définie dans Créature (cette propriété `Nom` est même définie en lecture et en écriture, ce qui est plus que le minimum requis par l'interface).

Dans la classe `Marchand`, ajoutez une méthode `AfficheOffre` qui affichera à la console le nom du marchand suivi de « propose » puis le nom de chaque compagnon disponible avec du prix d'achat et, s'il y a un coût d'entretien non nul, l'indication « plus … pièces d'or par semaine ». Par exemple :

```
Cunégonde propose :
(1) Abu : 20 po
(2) homme de main : 35 po plus 25 po par jour
(3) garde du corps : 84 po plus 35 po par jour
(4) vétéran : 140 po plus 45 po par jour
```

Étape 8 : des mercenaires

Dernière classe à ajouter : la classe `Mercenaire`, qui particularise la classe PNJ. Tout comme les familiers, les mercenaires sont des compagnons.

Le prix d'achat (solde à payer pour embaucher) d'un mercenaire dépend de son nombre de points de vie maximal : il s'agit de 7 pièces d'or par point de vie.

Le coût d'entretien d'un mercenaire dépend également de son nombre de points de vie maximal : jusqu'à 5 points de vie, le coût est de 25 po ; de 6 à 15 points de vie, le coût est de 35 po ; ensuite, on ajoute 2 po par point de vie au-delà du 15e.

(Note : ajoutez éventuellement une propriété en lecture seule à la classe `Créature` pour pouvoir calculer le prix d'achat et le coût d'entretien d'un mercenaire.)

Vous pourrez tester votre code en utilisant les lignes suivantes par exemple (elles pourraient nécessiter un petit ajout au code des classes). Le résultat affiché devrait correspondre à l'exemple donné à la fin de l'étape précédente.

```
Marchand cunégonde = new Marchand("Cunégonde");
cunégonde.AjouteCompagnon(abu);
Mercenaire m1 = new Mercenaire("homme de main", 5);
Mercenaire m2 = new Mercenaire("garde du corps", 12);
Mercenaire m3 = new Mercenaire("vétéran", 20);
cunégonde.AjouteCompagnon(m1, m2, m3);
cunégonde.AfficheOffre();
```