Laboratoire 5 : classes génériques

Objectifs

- Construire une classe générique
- Comprendre le fonctionnement et l'utilisation de classes et d'interfaces génériques
- Comprendre les restrictions sur les paramètres de types
- Utiliser des classes génériques prédéfinies
- Appliquer toutes ces connaissances dans le cadre d'exercices pratiques

Exercice 1 : Le todo d'appels téléphoniques

Certaines entreprises engagent des agences spécialisées dans la communication téléphonique ; ces dernières contactent des clients potentiels pour leur proposer divers articles. Dans cet exercice, vous construirez une classe C# destinée à gérer la liste des appels téléphoniques qu'un employé d'une telle agence doit effectuer.

Initialement, cette liste d'appels (qu'on appellera « todo ») sera vide, mais un superviseur pourra ajouter de nouveaux numéros à contacter. L'employé, quant à lui, devra appeler les numéros les uns après les autres.

(Pour la simplicité, on ne considère pas le fait que, lorsque l'interlocuteur ne répond pas ou raccroche trop rapidement, peut-être par énervement, l'employé est censé réessayer plus tard.)

Étape 1 : une première implémentation

Pour se simplifier la tâche, on considère que chaque numéro de téléphone est un simple entier.

Considérez l'implémentation proposée ci-dessous.

```
class Todo
{
  static private readonly int INCRÉMENT = 5;

  private int[] appels;
  private int nbAppels;

  public Todo ()
  {
    appels = new int[INCRÉMENT];
    nbAppels = 0;
  }

  public bool EstVide ()
  {
    return nbAppels == 0;
  }

  public void Ajoute (int appel)
  {
    if (nbAppels >= appels.Length)
    {
      int[] nvAppels = new int[appels.Length + INCRÉMENT];
      for (int i = 0; i < nbAppels; i++)
      {
```

```
          nvAppels[i] = appels[i];
        }
        appels = nvAppels;
      }
      appels[nbAppels] = appel;
      nbAppels++;
    }

    // Précondition : on n'appelle AppelLu que
    // dans le cas où le Todo n'est pas vide !
    public int AppelLu ()
    {
      int résultat = appels[0];
      int i = 1;
      while (i < nbAppels)
      {
        appels[i - 1] = appels[i];
        i++;
      }
      nbAppels--;
      return résultat;
    }
}
```

Lisez attentivement le code ; notez qu'il utilise le système des incréments présentés dans un laboratoire précédent pour ajuster la taille du tableau au nombre d'appels à mémoriser.

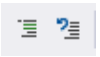
Testez son exécution via les instructions suivantes.

```
Todo todo = new Todo();
Console.WriteLine("Ajout de 111, 222, 333, 444, 555, 666");
todo.Ajoute(111);
todo.Ajoute(222);
todo.Ajoute(333);
todo.Ajoute(444);
todo.Ajoute(555);
todo.Ajoute(666);
Console.WriteLine("Lecture : " + todo.AppelLu());
Console.WriteLine("Lecture : " + todo.AppelLu());
Console.WriteLine("Ajout de 999");
todo.Ajoute(999);
while (!todo.EstVide())
{
  Console.WriteLine("Lecture : " + todo.AppelLu());
}
Console.WriteLine("Todo vide");
```

Étape 2 : une première modification

Les entiers ne sont pas forcément le meilleur choix pour représenter un numéro de téléphone. On décide donc d'utiliser des chaînes de caractères au lieu de simples entiers.

Faites un copier/coller du code de test actuel (pour qu'il apparaisse deux fois dans le programme principal). Sélectionnez une des deux copies et mettez-la intégralement en commentaires (raccourci clavier : Ctrl-K, Ctrl-C, C pour Comment ou le premier symbole des deux concernant les commentaires dans la barre d'outils). Cette copie en commentaires sera réutilisée plus tard.



Dans l'autre copie du code de test, celle qui n'est pas en commentaires, ajoutez des guillemets autour des nombres utilisés dans les 7 appels à `todo.Ajoute`. Par exemple, le premier appel devrait devenir comme suit.

```
todo.Ajoute("111");
```

Modifiez ensuite le code de la classe `Todo` pour qu'il accepte des chaînes de caractères. Cette modification devrait se limiter à remplacer certains des « `int` » par des « `string` » (plus exactement, vous devriez effectuer ce remplacement 7 fois). Nous espérons que vous avez employé le menu Edit ou toute autre raccourci tel que CTRL H.

Vérifiez bien que l'exécution du code test donne le même résultat que précédemment.

Étape 3 : une seconde modification

Les chaînes de caractères, ce n'est pas mal… mais on peut faire mieux.

Définissez une nouvel classe appelée `Appel`. Chaque objet de cette classe sera caractérisé par deux chaînes de caractères : un nom et un numéro de téléphone. Ajoutez également à la classe une méthode `ToString` présentant l'objet sous la forme d'une chaîne constituée du nom suivi du numéro de téléphone entre parenthèses.

À nouveau, commencez par modifier le code de test. Sélectionnez l'ensemble des instructions qui vous ont permis de tester la version « chaîne de caractères » de l'étape précédente et faites-en une copie. Mettez ensuite en commentaires l'une des deux copies.

Modifiez l'autre copie en y ajoutant les définitions ci-dessous et en remplaçant dans chacun des appels à Ajoute l'argument par un argument du type « a » suivi d'un numéro. Ainsi, le premier appel devrait devenir `todo.Ajoute(a1)`.

```
Appel a1 = new Appel("Ilove Ones", "111");
Appel a2 = new Appel("Parité Forever", "222");
Appel a3 = new Appel("Demy Daemon", "333");
Appel a4 = new Appel("Poule Chicken", "444");
Appel a5 = new Appel("Phone Number", "555");
Appel a6 = new Appel("L. A. Bête", "666");
Appel a9 = new Appel("Triple Ponte", "999");
```

Modifiez ensuite le code de la classe Todo pour qu'il accepte désormais des objets de type `Appel` au niveau de vos méthodes (7 remplacements en tout).

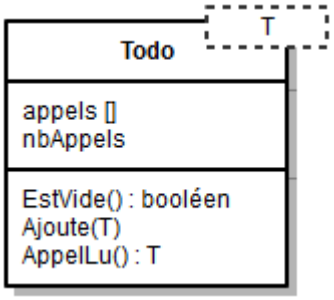
Cette fois-ci, l'exécution du code test devrait produire l'affichage suivant.

```
Ajout de 111, 222, 333, 444, 555, 666
Lecture : Ilove Ones (111)
Lecture : Parité Forever (222)
Ajout de 999
Lecture : Demy Daemon (333)
Lecture : Poule Chicken (444)
Lecture : Phone Number (555)
Lecture : L. A. Bête (666)
Lecture : Triple Ponte (999)
Todo vide
```

Étape 4 : un code générique

Les trois versions précédentes de la classe `Todo` utilisent fondamentalement le même code. En effet, les seules différences sont les 7 types, qui étaient « `int` » dans la première version, « `string` » dans la 2e version et « `Appel` » dans la 3e version.

Pour pouvoir utiliser un même code avec plusieurs types, on peut utiliser une classe générique.



En pratique, cela revient à utiliser une sorte de « variable de type » `T` qui pourra être tantôt « `int` », tantôt « `string` », tantôt « `Appel` » ou un autre type (cf. la représentation UML d'une classe générique).

Modifiez le code de la classe `Todo` comme suit :

1. Remplacez le nom de la classe, `Todo`, par `Todo<T>`.
2. Dans la classe, remplacez les occurrences de `Appel` (celles qui avaient été `int` puis `string` auparavant) par `T`.

La classe `Todo` est désormais une classe générique. Pour pouvoir l'utiliser, il faut tout d'abord préciser le type qu'on utilise (en quelque sorte la « valeur » de la « variable » `T`). Cela se fait en utilisant la syntaxe suivante (que vous pouvez recopier dans le code test).

```
Todo<Appel> todo = new Todo<Appel>();
```

Vérifiez que le code test donne le bon résultat.

Placez ensuite en commentaire le code test correspondant à la version « Appel » (celui que vous venez d'exécuter) et sortez des commentaires celui qui correspond à la version « string » (sélection puis raccourci clavier Ctrl-K Ctrl-U, U pour Uncomment ou l'onglet prédéfini dans la barre d'outils). Dans celui-ci, modifiez la déclaration de la variable `todo` en la ligne suivante.

```
Todo<string> todo = new Todo<string>();
```

Exécutez le code test… et observez que le code générique de la classe `Todo<T>` est compatible avec le type `string`.

Finalement, décommentez le code test correspondant à la version utilisant des entiers. Modifiez à nouveau la déclaration de la variable `todo` pour cette version fonctionne avec le code générique.

Étape 5 : suppression des répétitions

Fondamentalement, on n'a pas vraiment inventé quelque chose de nouveau en codant la classe « Todo ». Il s'agit tout simplement d'une structure standard fort connue en informatique. Voyez-vous laquelle ? (Indice : souvenez-vous de votre cours d'OED).

On peut toutefois améliorer un peu la structure utilisée ici dans le cas particulier des listes d'appels téléphoniques, par exemple en évitant de placer deux fois le même numéro d'appel dans la liste. Concrètement, cela reviendra à tester qu'un appel n'est pas déjà présent dans le todo quand on demande de l'ajouter.

Vérifier le fait qu'un appel téléphonique se trouve dans la liste des appels est une activité à part entière qui mérite sa propre méthode (plutôt que d'être enfouie dans le code de la méthode `Ajoute`). Codez donc une méthode privée `Contient` qui indique (par un booléen) si le todo contient déjà un appel donné.

Normalement, Visual Studio devrait indiquer une erreur dans votre code… Lisez attentivement le message d'erreur, qui devrait être quelque chose comme

```
Operator '!=' cannot be applied to operands of type 'T' and 'T'
```

Comme le message l'indique, l'erreur vient du fait qu'on ne peut pas être certain que le test d'égalité (ou test d'inégalité) soit utilisable avec les objets de type `T`.

Pour pouvoir tester l'égalité entre deux éléments de type `T` (et ainsi vérifier qu'un appel se trouve déjà dans le todo), il faut imposer une restriction sur le type `T` : on n'autorise que les types `T` dont on peut comparer les éléments.

De manière générale, demander qu'une classe ou qu'un type possède certaines opérations (comme le test d'égalité) se traduit par une contrainte de la forme « la classe/le type doit implémenter une interface donnée. » Dans ce cas-ci, on va utiliser l'interface prédéfinie `IComparable`.

Comme on peut le voir sur sa page de documentation (https://msdn.microsoft.com/en-us/library/system.icomparable(v=vs.110).aspx), cette interface ne comporte qu'une méthode, `int CompareTo(Object)`, dont l'implémentation doit correspondre à la condition suivante :

> `o.CompareTo(obj)` = 0 si et seulement `o` et `obj` sont équivalents/égaux[1].

Pour ajouter cette contrainte sur le type T, il faut remplacer l'en-tête de la classe Todo par la ligne suivante

[1] Notez que la spécification de `CompareTo` est un peu plus complexe que cela vu qu'elle permet aussi d'indiquer si `o` est plus « petit » ou plus « grand » que `obj`, mais ce n'est pas pertinent dans le cadre de cet exercice.

```
class Todo<T> where T : IComparable
```

ce qui permet alors de tester l'égalité entre deux « appels » de type T via `appel1.CompareTo(appel2) == 0`.

Modifiez votre code en conséquence, puis utilisez la nouvelle méthode `Contient` pour faire en sorte qu'on n'ajoute un appel au todo que s'il n'est pas encore présent.

Pour tester votre code, dupliquez par exemple les lignes

```
todo.Ajoute(111);
todo.Ajoute(222);
```

dans le test et vérifiez que les numéros 111 et 222 n'apparaissent pas deux fois.

Étape 6 : IComparable

Le code test a normalement fonctionné immédiatement avec la version « `int` ». En fait, l'interface IComparable est implémentée par la plupart des types et classes prédéfinis.

En commentant/décommentant les lignes adéquates, testez que c'est bien le cas du type « `string` » également (à nouveau, dupliquez certaines lignes d'ajout de numéros de téléphone pour vérifier que les doublons ne sont pas ajoutés).

Finalement, décommentez uniquement le code test relatif à la version « Appel » et tentez de l'exécuter avec la nouvelle version de `Todo`. Cela devrait produire une erreur à la compilation. Assurez-vous de bien lire le message d'erreur indiqué, que vous devriez être capables de comprendre !

Étape 7 : rendre Appel IComparable

Pour pouvoir continuer à utiliser la classe Appel comme type T, il faut s'assurer qu'elle implémente l'interface IComparable. Cela signifie qu'il faut entre autres lui ajouter une méthode `public int CompareTo (Object o)`.

Notez bien que l'argument est, dans ce cas-ci, un objet quelconque (donc pas forcément un `Appel`). Voici comment cette méthode devrait procéder :

1. On teste si o contient une référence.
2. Si l'objet `o` n'est pas une instance d'Appel, on renvoie d'office -1 pour indiquer que l'objet courant (« `this` ») et `o` ne sont pas équivalents.
3. Sinon, si c'est bien une instance d'`Appel`, on vérifiera uniquement que les numéros de téléphone correspondent (en ignorant les noms : après tout, si deux noms sont associés au même numéro de téléphone, on désire ne passer qu'un seul appel !).

Pour tester si `o` est une instance de `Appel` (ce qui se noterait `o instanceof Appel` en Java), C# utilise la syntaxe `o is Appel`. Si `o` est bel et bien une instance de `Appel`, on peut le « caster » (en français correct, « transtyper ») en un objet de type Appel en utilisant la syntaxe usuelle, à savoir `(Appel)o`. C'est seulement après cela qu'on pourra accéder aux attributs de l'objet en question.

Notez que le C# propose également une autre syntaxe pour vérifier si un objet est une instance d'un autre type : `o as Appel`. Cette expression renvoie un objet qui est soit `(Appel)o` si `o` est bien une instance de `Appel`, soit `null` dans le cas contraire.

Utilisez l'une ou l'autre méthode puis testez votre implémentation en ajoutant par exemple les lignes suivantes dans le code test pour vérifier que le numéro 333 n'est pas ajouté deux fois.

```
Appel a3bis = new Appel("Triple trois", "333");
todo.Ajoute(a3bis);
```

Étape 8 : un note sur IComparable<T>

À côté de l'interface `IComparable` utilisée plus haut (et qui impose une méthode de comparaison avec un objet de type `Object`), il existe une interface générique `IComparable<T>` qui impose une méthode de comparaison avec un objet de type `T`.

On aurait pu utiliser la restriction `where T : IComparable<T>` plus haut afin de ne devoir implémenter qu'une comparaison entre appels (ce qui nous aurait évité le casting d'`Object` en `Appel`), mais c'était une bonne occasion d'introduire les opérateurs `is` et `as`.

Étape 9 : aller plus loin (dépassement)

L'implémentation choisie pour un todo n'est vraiment pas la plus optimale. Un de ces principaux défauts est le fait que toutes les valeurs du todo doivent être déplacées chaque fois qu'on lit un appel. Ces déplacements ennuyeux sont nécessaires parce qu'on désire que le prochain appel à lire se trouve toujours dans la première cellule du tableau.

Si on enlève cette restriction, on peut obtenir une implémentation plus efficace mais, en contrepartie, il faut retenir l'indice de la cellule où se trouve le prochain appel à lire (dans un attribut appelé `iDébut`) ainsi que l'indice de fin (`iFin`).

À l'initialisation (et chaque fois que le todo sera vide), on placera par convention la valeur `-1` dans `iDébut`. Dans les autres cas, voici comment ces indices vont évoluer.

Voici le type de situation à laquelle on arriverait après l'ajout de plusieurs numéros.

111	222	333	444	
↑iDébut			↑iFin	

Si on demande d'effectuer une lecture à ce moment-là, l'appel renvoyé est 111 et la structure en mémoire devient la suivante.

	222	333	444	
	↑iDébut		↑iFin	

Une seconde lecture produirait l'appel 222 et le résultat suivant.

		333	444	
		↑iDébut	↑iFin	

À ce moment-là, si on demandait l'ajout des numéros 555 puis 666, la valeur `iFin` atteindrait la dernière case du tableau puis repartirait au début pour occuper les premières cellules qui sont libres.

666		333	444	555
↑iFin		↑iDébut		

À ce moment-ci, on peut encore ajouter un numéro (car il reste une cellule libre). Ensuite, on se rend compte que le tableau est plein car, si on avance `iFin`, il rejoint `iDébut`. On se rend donc compte qu'il faut agrandir le tableau en ajoutant un incrément. Comme, de toute façon, il est nécessaire de recopier toutes les valeurs dans le nouveau tableau, on en profite pour les remettre dans l'ordre des cellules. Voici les résultats qu'on obtiendrait si on ajoutait les numéros 777 puis 888 (en supposant un incrément de 3).

666	777	333	444	555
	↑iFin	↑iDébut		

333	444	555	666	777	888		
↑iDébut					↑iFin		

Si vous désirez affronter un problème d'algorithmique plus complexe que d'habitude, vous pouvez tenter de modifier le code de la classe `Todo<T>` pour qu'il utilise ce procédé.

Exercice 2 : Les services secrets

L'objectif de cet exercice est d'utiliser divers types génériques prédéfinis et leurs méthodes en se basant sur la documentation en ligne. Le décor de l'exercice est celui de services secrets possédant des chapitres dans diverses villes ainsi que plusieurs agents secrets qui sont en contact avec des informateurs.

Note. À la fin de cet exercice, vous trouverez un exemple de code permettant de générer un objet de la classe Chapitre. Vous pouvez puiser dans ce code pour obtenir des exemples lorsque vous testerez les méthodes que vous aurez écrites.

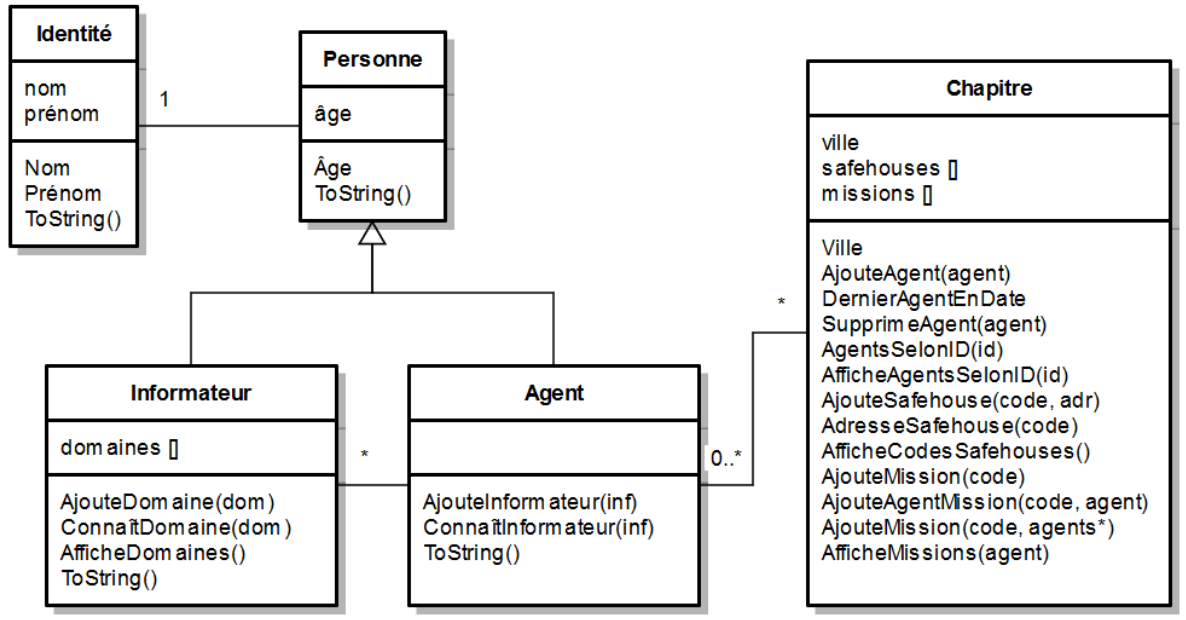
Limite : nous n'écrirons pas toutes les vérifications. Exemple : AjouteDomaine qui ajoute un domaine devrait vérifier que le domaine est non null.

Étape 1 : les classes de base

Commencez par créer les classes `Identité` et `Personne` du schéma UML ci-dessous.

Une `Identité` est un objet constitué d'un nom (de famille) et d'un prénom. Prévoyez des propriétés publiques pour accéder à ces valeurs en lecture ainsi qu'un constructeur standard.

Une `Personne` possédera une identité et un âge. Là aussi, prévoyez des propriétés permettant un accès en lecture (ou utilisez des propriétés auto-implémentées).



Étape 2 : informateurs

La classe `Informateur` est une sous-classe de `Personne`.

En plus de son identité et de son âge, un informateur possède des domaines de connaissances (par exemple : « pègre », « drogue », « vente d'armes », « contrebande »). Ces domaines de connaissances seront modélisés sous la forme d'une collection de chaînes de caractères.

Dans ce cas-ci, utilisez le type générique `HashSet<T>` pour l'attribut privé `domaines`. Consultez la documentation en ligne (une recherche sur les termes « C# HashSet<T> » devrait vous mener à bon port) pour connaître le nom des méthodes disponibles.

Définissez les trois méthodes suivantes dans la classe `Informateur` :

- une méthode `AjouteDomaine` permettant d'ajouter un domaine à la liste des domaines existants (lisez la spécification de la méthode Add et plus particulièrement la valeur qu'elle renvoie pour bien comprendre son fonctionnement — notez que la valeur qu'elle renvoie ne nous est pas utile dans ce cas précis),
- une méthode `ConnaîtDomaine` indiquant si l'informateur connaît le domaine donné en paramètre (et renvoyant donc un booléen), et
- une méthode `AfficheDomaines` qui affiche la liste des domaines de connaissances de l'utilisateur sur la console (principalement pour les tests) ; notez que la plupart des collections (c'est le cas de `HashSet<T>`) sont énumérables, c'est-à-dire qu'elles peuvent être utilisées avec une boucle `foreach` (ce qui sera utile ici).

Testez ces trois méthodes, et plus particulièrement le cas de l'ajout d'un domaine de connaissance qui existe déjà.

Que se passe-t-il si vous ajoutez successivement les domaines de connaissances « contrebande » puis « Contrebande » à un informateur ? Pour éviter ce genre de problèmes, assurez-vous que toutes les chaînes de caractères stockées dans la collection `domaines` soient écrites intégralement en minuscules. Adaptez le code des méthodes pour qu'elles fonctionnent correctement quelle que soit la manière dont le domaine donné en paramètre est écrit.

Étape 3 : agents secrets

Un `Agent` est un type particulier de `Personne`. Un agent possède une liste de contacts qui sont des informateurs (liste qui sera initialement vide à la création d'un agent). Codez cette liste en utilisant à nouveau la classe `HashSet<T>`.

Ici encore, ajoutez des méthodes permettant d'ajouter un nouvel informateur (si l'informateur est déjà cité, on ne l'ajoute pas) et indiquant si un informateur donné fait partie des contacts de l'agent.

Étape 4 : chapitres

Les services secrets gèrent des chapitres établis dans diverses villes.

Pour chaque `Chapitre`, on retient

- le nom de la ville où le chapitre est établi (vous pouvez utiliser une propriété auto-implémentée) ;
- la liste des agents liés à ce chapitre encodés par ordre d'ancienneté décroissante au sein du chapitre ; et
- la liste des adresses (chaînes de caractères) des bâtiments sécurisés où un agent peut se replier en cas de problèmes, chacun d'entre eux étant repéré par un nom de code ;
- la liste des missions en cours dans ce chapitre avec, pour chacune, son code alphanumérique et les agents impliqués.

Chacun de ces attributs (à part le nom de la ville) est abordé dans les étapes qui suivent.

Prévoyez un constructeur qui crée un nouveau chapitre sans contenu dans les collections (à mettre à jour au fur et à mesure de l'ajout des attributs-collections).

Étape 5 : liste des agents d'un chapitre

Pour la liste des agents associés à un chapitre donné, on pourrait utiliser `HashSet<T>` si l'ordre dans lequel les agents sont encodés n'avait pas d'importance. Si on désire conserver l'ordre d'encodage et faciliter l'implémentation d'opérations telles que la recherche de l'agent « suivant » ou de l'agent « précédent », on peut préférer utiliser une collection de type `LinkedList<T>`.

Dans un premier temps, recherchez la documentation en ligne au sujet de ce type de collections qui implémente une liste chaînée double (chaque cellule de la liste contient des informations permettant de retrouver celui qui suit et celui qui précède).

Prévoyez les méthodes suivantes.

- `AjouteAgent(agent)` permettra d'ajouter un nouvel agent qui sera placé automatiquement à la fin de la liste (mais seulement si l'agent en question n'est pas déjà présent dans la liste).
- La propriété `DernierAgentEnDate` donnera, en lecture, l'agent le plus récent de la liste (ou null si la liste est vide). En écriture, il permettra d'ajouter un nouvel agent en fin de liste (comme `AjouteAgent`).
- La méthode `SupprimeAgent(agent)` permettra d'enlever un agent de la liste. Elle ne fera rien du tout si l'agent indiqué ne se trouve pas dans la liste.
- La méthode `AgentsSelonID(identité)` recherchera, dans la liste des agents, ceux dont l'identité correspondent à celle donnée en paramètre. Si, dans l'identité donnée, l'attribut nom vaut `null`, cela signifie qu'on n'impose aucune contrainte sur le nom ; s'il ne vaut pas `null`, on ne s'intéresse qu'aux agents portant ce nom (idem pour le prénom). Pour implémenter cette méthode, utilisez une boucle `foreach` qui construira petit à petit le résultat sous la forme d'un `HashSet` d'agents. (Note : une autre méthode sera proposée plus tard.)
- La méthode `AfficheAgentsSelonID(identité)` utilisera la précédente et affichera la liste des agents correspondant à l'identité donnée (pour faciliter les tests).

Étape 6 : safehouses et leur nom de code

Chaque adresse de maisons sécurisées (ou « safehouses ») est désignée par un nom de code qui prend la forme d'une chaîne de caractères.

Ici, il s'agit donc de stocker une liste de noms de code avec, pour chacun, l'adresse qui correspond. Cela correspond à une structure de tableau associatif (voir cours de Javascript) ou encore de dictionnaire où les clefs sont les noms de code et les valeurs, les adresses.

Pour l'attribut `safehouses`, utilisez la classe générique. Il s'agit d'une classe générique à deux paramètres de type : le type des clefs, `TKey`, et le type de valeurs, `TValue`. Dans le

cas de safehouses, les clefs et les valeurs sont des chaînes de caractères (mais les deux types pourraient être différents).

Prévoyez les méthodes suivantes.

- `AjouteSafehouse(code, adresse)` ajoutera une nouvelle safehouse à la liste. Lisez bien la spécification de la méthode `Add` dans la documentation en ligne. Cette méthode peut soulever deux types d'exception (nous n'aborderons pas les exceptions dans ce cours). Mais assurez-vous qu'un code déjà existant ne peut être ajouté !
- `AdresseSafehouse(code)` renverra soit « Code inconnu » soit l'adresse de la safehouse correspondant au code donné. Lisez bien la documentation de la méthode `TryGetValue` pour l'utiliser correctement !
- `AfficheCodesSafehouses()` affichera dans la console la liste des codes des safehouses. Pour parcourir le dictionnaire, utilisez un foreach avec la syntaxe suivante (où on utilise une variable implicitement typée : https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/var)

```
foreach(var safehouse in safehouses)
{
…
}
```

Note. La classe `Dictionary<TKey,TValue>` possède une propriété au nom peut-être un peu surprenant, à savoir `Item[TKey]`. Dans la définition d'une classe, C# permet de placer une méthode indiquant ce qui se produit si on utilise une instance de la classe comme un tableau, avec une syntaxe du genre `obj[index]`. Dans le cas d'un dictionnaire, la syntaxe `dict[clef]` permet d'accéder directement (en lecture ou en écriture) à la valeur associée à la clef indiquée. Ainsi, on peut lire cette valeur via par exemple `dict["code"]` ou encore ajouter une nouvelle valeur / modifier une valeur existante via une affectation `dict["code"] = "adresse safehouse"` dans notre cas (notez au passage le comportement différent de Add au cas où la clef existe déjà).

Étape 7 : missions et les agents associés

Un chapitre peut être en charge de missions. Chaque mission comporte un nom de code (chaîne de caractères) et est attribuée à un ou à plusieurs agents (associés ou chapitre ou pas).

L'attribut `missions` sera implémentée en utilisant également `Dictionary<TKey,TValue>` mais, dans ce cas-ci, le type `TValue` (c'est-à-dire le type des informations associées à un nom de code) sera un ensemble d'agents. Pour cet ensemble, on utilisera `HashSet<T>`. À vous de combiner toutes ces informations pour déclarer l'attribut en question et pour définir les méthodes suivantes.

- `AjouteMission(code)` ajoute une nouvelle mission sans aucun agent associé (pour le moment) ; s'il existe déjà une mission correspondant au code donné, on ne fait rien.
- `AjouteAgentMission(code, agent)` ajout l'agent cité à la liste des agents chargés de la mission correspondant au code donné (si la mission n'existe pas, elle est créée).

- `AjouteMission(code, agent1, agent2, …, agentn)` ajoute l'information indiquant que les agents cités sont affectés à la mission indiquée. Si la mission n'existe pas encore, elle est créée ; si elle existait déjà, on ajoute tout simplement les agents.
- `AfficheMissions(agent)` affiche le code de toutes les missions qui concerne l'agent donné.

Note. Si vous testez la dernière méthode avec l'exemple donné en fin d'exercice, l'appel `c1.AfficheMissions(c1m0a1);` *devrait citer les missions Chicorée, Clématite et Joubarde.*

Étape 8 : des toString

Ajoutez aux diverses classes que vous avez définies jusqu'ici des méthodes `toString`. Arrangez-vous pour que, dans le cas d'un chapitre, le résultat produit corresponde au format de l'exemple suivant. (Note : pour les autres classes, choisissez un format qui vous facilitera la construction de la sortie pour `Chapitre`.)

```
INFORMATIONS SUR LE CHAPITRE DE MANAGE

AGENTS :

Natalie, Mills (26)
 - Carter, Howard (18) [crime organisé, espionnage technique]
 - Grace, Potter (33) [crime organisé, espionnage technique, vente
d'armes]
 - Evelyn, Cox (41) [crime organisé]

Peyton, Davidson (23)
 - Aiden, Hopkins (47) [mafia]
 - Kaylee, Owens (26) [cambriolages, vente d'armes]

SAFEHOUSES

Genêt : Ruelle Marique, 125
Astrance : Chemin de Bateau, 234
Buglose : Rue Tabaral, 357
Saponaire : Chemin Masy, 128

MISSION Chicorée (2 agents)

Isaiah, Island (28)
 - Daniel, Mc (18) [drogue, pègre]
 - Mason, Jackson (22) [drogue, mafia, contrebande]
 - Addison, Phillips (39) [espionnage technique]

Lincoln, Bros (42)
 - Elena, Watts (31) [vente d'armes, cambriolages]
 - Grace, Bryan (39) [mafia]
```

```
MISSION Lys (3 agents)

Addison, Reynolds (43)
 - Charlotte, Hunt (40) [drogue]
 - Aiden, Bryan (20) [drogue, racket, mafia]
 - Ella, Fletcher (29) [cambriolages]
 - Hailey, Fletcher (31) [pègre, blanchiment d'argent]
 - Scarlett, Lee (37) [cambriolages]
 - Emily, Roberts (40) [contrebande, blanchiment d'argent]

Hailey, Graham (20)
 - Jayce, Nichols (30) [espionnage technique]
 - Oliver, Wood (20) [contrebande, crime organisé]

Hailey, Hill (39)
 - Mackenzie, Kerr (33) [crime organisé]
 - Addison, Knight (25) [mafia, blanchiment d'argent, pègre]
 - Nicholas, Hunter (30) [mafia]
 - Nicholas, Davis (28) [blanchiment d'argent]
 - Victoria, Kerr (44) [mafia]
 - Landon, Ellis (22) [mafia, pègre, racket]
```

Exemple de code définissant un chapitre

Les instructions suivantes définissent un chapitre nommé `c1`. Notez que ces lignes ont été produites par un générateur automatique dans le but de fournir un exemple complet, et que les noms de variables (générés eux aussi automatiquement) ne sont donc pas des exemples à suivre !

```
Chapitre c1 = new Chapitre("Manage");
Agent c1aa0 = new Agent(new Identité("Natalie", "Mills"), 26);
Informateur c1aa0i0 = new Informateur(new Identité("Carter", "Howard"),
18);
c1aa0i0.AjouteDomaine("crime organisé");
c1aa0i0.AjouteDomaine("espionnage technique");
c1aa0.AjouteInformateur(c1aa0i0);
Informateur c1aa0i1 = new Informateur(new Identité("Grace", "Potter"),
33);
c1aa0i1.AjouteDomaine("crime organisé");
c1aa0i1.AjouteDomaine("espionnage technique");
c1aa0i1.AjouteDomaine("vente d'armes");
c1aa0.AjouteInformateur(c1aa0i1);
Informateur c1aa0i2 = new Informateur(new Identité("Evelyn", "Cox"), 41);
c1aa0i2.AjouteDomaine("crime organisé");
c1aa0.AjouteInformateur(c1aa0i2);
c1.AjouteAgent(c1aa0);
```

```
Agent c1aa1 = new Agent(new Identité("Peyton", "Davidson"), 23);
Informateur c1aa1i0 = new Informateur(new Identité("Aiden", "Hopkins"),
47);
c1aa1i0.AjouteDomaine("mafia");
c1aa1.AjouteInformateur(c1aa1i0);
Informateur c1aa1i1 = new Informateur(new Identité("Kaylee", "Owens"),
26);
c1aa1i1.AjouteDomaine("cambriolages");
c1aa1i1.AjouteDomaine("vente d'armes");
c1aa1.AjouteInformateur(c1aa1i1);
c1.AjouteAgent(c1aa1);
Agent c1aa2 = new Agent(new Identité("Landon", "Alexander"), 30);
Informateur c1aa2i0 = new Informateur(new Identité("Isaac", "Parker"),
39);
c1aa2i0.AjouteDomaine("espionnage technique");
c1aa2i0.AjouteDomaine("mafia");
c1aa2i0.AjouteDomaine("blanchiment d'argent");
c1aa2.AjouteInformateur(c1aa2i0);
Informateur c1aa2i1 = new Informateur(new Identité("Benjamin", "Duncan"),
22);
c1aa2i1.AjouteDomaine("racket");
c1aa2i1.AjouteDomaine("vente d'armes");
c1aa2.AjouteInformateur(c1aa2i1);
Informateur c1aa2i2 = new Informateur(new Identité("Mason", "Miss"), 34);
c1aa2i2.AjouteDomaine("crime organisé");
c1aa2i2.AjouteDomaine("drogue");
c1aa2i2.AjouteDomaine("vente d'armes");
c1aa2.AjouteInformateur(c1aa2i2);
c1.AjouteAgent(c1aa2);
Agent c1aa3 = new Agent(new Identité("Hannah", "Mary"), 47);
Informateur c1aa3i0 = new Informateur(new Identité("Hannah", "Murphy"),
27);
c1aa3i0.AjouteDomaine("drogue");
c1aa3i0.AjouteDomaine("contrebande");
c1aa3i0.AjouteDomaine("racket");
c1aa3.AjouteInformateur(c1aa3i0);
Informateur c1aa3i1 = new Informateur(new Identité("Brooklyn", "Page"),
25);
c1aa3i1.AjouteDomaine("pègre");
c1aa3i1.AjouteDomaine("pègre");
c1aa3i1.AjouteDomaine("espionnage technique");
c1aa3.AjouteInformateur(c1aa3i1);
c1.AjouteAgent(c1aa3);
Agent c1aa4 = new Agent(new Identité("Julian", "Hicks"), 38);
Informateur c1aa4i0 = new Informateur(new Identité("Liam", "Murray"),
31);
c1aa4i0.AjouteDomaine("mafia");
```

```
c1aa4.AjouteInformateur(c1aa4i0);
Informateur c1aa4i1 = new Informateur(new Identité("Christian",
"Thomas"), 42);
c1aa4i1.AjouteDomaine("blanchiment d'argent");
c1aa4i1.AjouteDomaine("mafia");
c1aa4i1.AjouteDomaine("mafia");
c1aa4.AjouteInformateur(c1aa4i1);
c1.AjouteAgent(c1aa4);
c1.AjouteSafehouse("Genêt", "Ruelle Marique, 125");
c1.AjouteSafehouse("Astrance", "Chemin de Bateau, 234");
c1.AjouteSafehouse("Buglose", "Rue Tabaral, 357");
c1.AjouteSafehouse("Saponaire", "Chemin Masy, 128");
c1.AjouteSafehouse("Digitale", "Chemin de la Taille Boha, 331");
c1.AjouteSafehouse("Myosotis", "Rue Quirin, 142");
c1.AjouteSafehouse("Potentille", "Place Sainte-Sévère, 277");
Agent c1m0a0 = new Agent(new Identité("Isaiah", "Island"), 28);
Informateur c1m0a0i0 = new Informateur(new Identité("Daniel", "Mc"), 18);
c1m0a0i0.AjouteDomaine("drogue");
c1m0a0i0.AjouteDomaine("pègre");
c1m0a0.AjouteInformateur(c1m0a0i0);
Informateur c1m0a0i1 = new Informateur(new Identité("Mason", "Jackson"),
22);
c1m0a0i1.AjouteDomaine("drogue");
c1m0a0i1.AjouteDomaine("mafia");
c1m0a0i1.AjouteDomaine("contrebande");
c1m0a0.AjouteInformateur(c1m0a0i1);
Informateur c1m0a0i2 = new Informateur(new Identité("Addison",
"Phillips"), 39);
c1m0a0i2.AjouteDomaine("espionnage technique");
c1m0a0.AjouteInformateur(c1m0a0i2);
c1.AjouteAgentMission("Chicorée", c1m0a0);
Agent c1m0a1 = new Agent(new Identité("Lincoln", "Bros"), 42);
Informateur c1m0a1i0 = new Informateur(new Identité("Elena", "Watts"),
31);
c1m0a1i0.AjouteDomaine("vente d'armes");
c1m0a1i0.AjouteDomaine("cambriolages");
c1m0a1i0.AjouteDomaine("cambriolages");
c1m0a1.AjouteInformateur(c1m0a1i0);
Informateur c1m0a1i1 = new Informateur(new Identité("Grace", "Bryan"),
39);
c1m0a1i1.AjouteDomaine("mafia");
c1m0a1.AjouteInformateur(c1m0a1i1);
c1.AjouteAgentMission("Chicorée", c1m0a1);
Agent c1m1a0 = new Agent(new Identité("Daniel", "Williamson"), 34);
Informateur c1m1a0i0 = new Informateur(new Identité("Noah", "Robert"),
22);
c1m1a0i0.AjouteDomaine("espionnage technique");
```

```
c1m1a0.AjouteInformateur(c1m1a0i0);
Informateur c1m1a0i1 = new Informateur(new Identité("William", "Clay"),
45);
c1m1a0i1.AjouteDomaine("drogue");
c1m1a0i1.AjouteDomaine("cambriolages");
c1m1a0i1.AjouteDomaine("drogue");
c1m1a0.AjouteInformateur(c1m1a0i1);
Informateur c1m1a0i2 = new Informateur(new Identité("Grayson", "Perry"),
31);
c1m1a0i2.AjouteDomaine("blanchiment d'argent");
c1m1a0.AjouteInformateur(c1m1a0i2);
c1.AjouteAgentMission("Gentiane", c1m1a0);
Agent c1m1a1 = new Agent(new Identité("Nora", "Bruce"), 24);
Informateur c1m1a1i0 = new Informateur(new Identité("Oliver", "Sherman"),
47);
c1m1a1i0.AjouteDomaine("mafia");
c1m1a1i0.AjouteDomaine("espionnage technique");
c1m1a1i0.AjouteDomaine("vente d'armes");
c1m1a1.AjouteInformateur(c1m1a1i0);
Informateur c1m1a1i1 = new Informateur(new Identité("Liam", "Stanley"),
33);
c1m1a1i1.AjouteDomaine("mafia");
c1m1a1.AjouteInformateur(c1m1a1i1);
Informateur c1m1a1i2 = new Informateur(new Identité("Gabriel", "Watts"),
47);
c1m1a1i2.AjouteDomaine("racket");
c1m1a1i2.AjouteDomaine("espionnage technique");
c1m1a1.AjouteInformateur(c1m1a1i2);
Informateur c1m1a1i3 = new Informateur(new Identité("Evelyn", "Page"),
33);
c1m1a1i3.AjouteDomaine("blanchiment d'argent");
c1m1a1i3.AjouteDomaine("cambriolages");
c1m1a1i3.AjouteDomaine("drogue");
c1m1a1.AjouteInformateur(c1m1a1i3);
Informateur c1m1a1i4 = new Informateur(new Identité("Owen", "Fleming"),
44);
c1m1a1i4.AjouteDomaine("blanchiment d'argent");
c1m1a1i4.AjouteDomaine("vente d'armes");
c1m1a1.AjouteInformateur(c1m1a1i4);
Informateur c1m1a1i5 = new Informateur(new Identité("Sophia", "Dixon"),
23);
c1m1a1i5.AjouteDomaine("pègre");
c1m1a1i5.AjouteDomaine("espionnage technique");
c1m1a1.AjouteInformateur(c1m1a1i5);
c1.AjouteAgentMission("Gentiane", c1m1a1);
Agent c1m2a0 = new Agent(new Identité("Isaiah", "Crawford"), 34);
Informateur c1m2a0i0 = new Informateur(new Identité("Dylan", "Lee"), 25);
```

```
c1m2a0i0.AjouteDomaine("crime organisé");
c1m2a0.AjouteInformateur(c1m2a0i0);
Informateur c1m2a0i1 = new Informateur(new Identité("Abigail", "Morgan"),
20);
c1m2a0i1.AjouteDomaine("drogue");
c1m2a0.AjouteInformateur(c1m2a0i1);
Informateur c1m2a0i2 = new Informateur(new Identité("Carter", "Kelly"),
41);
c1m2a0i2.AjouteDomaine("contrebande");
c1m2a0i2.AjouteDomaine("drogue");
c1m2a0.AjouteInformateur(c1m2a0i2);
Informateur c1m2a0i3 = new Informateur(new Identité("Andrew", "Bell"),
35);
c1m2a0i3.AjouteDomaine("blanchiment d'argent");
c1m2a0i3.AjouteDomaine("cambriolages");
c1m2a0i3.AjouteDomaine("espionnage technique");
c1m2a0.AjouteInformateur(c1m2a0i3);
c1.AjouteAgentMission("Clématite", c1m2a0);
c1.AjouteAgentMission("Clématite", c1m0a1);
Agent c1m2a1 = new Agent(new Identité("Lincoln", "Graham"), 31);
Informateur c1m2a1i0 = new Informateur(new Identité("Emily", "May"), 24);
c1m2a1i0.AjouteDomaine("blanchiment d'argent");
c1m2a1i0.AjouteDomaine("racket");
c1m2a1.AjouteInformateur(c1m2a1i0);
Informateur c1m2a1i1 = new Informateur(new Identité("Layla",
"Washington"), 34);
c1m2a1i1.AjouteDomaine("drogue");
c1m2a1i1.AjouteDomaine("drogue");
c1m2a1i1.AjouteDomaine("pègre");
c1m2a1.AjouteInformateur(c1m2a1i1);
Informateur c1m2a1i2 = new Informateur(new Identité("Mason", "Harrison"),
37);
c1m2a1i2.AjouteDomaine("pègre");
c1m2a1i2.AjouteDomaine("mafia");
c1m2a1.AjouteInformateur(c1m2a1i2);
Informateur c1m2a1i3 = new Informateur(new Identité("Dylan", "Blake"),
33);
c1m2a1i3.AjouteDomaine("vente d'armes");
c1m2a1i3.AjouteDomaine("blanchiment d'argent");
c1m2a1i3.AjouteDomaine("cambriolages");
c1m2a1.AjouteInformateur(c1m2a1i3);
Informateur c1m2a1i4 = new Informateur(new Identité("Adeline", "Ann"),
35);
c1m2a1i4.AjouteDomaine("espionnage technique");
c1m2a1i4.AjouteDomaine("drogue");
c1m2a1.AjouteInformateur(c1m2a1i4);
```

```
Informateur c1m2a1i5 = new Informateur(new Identité("Michael", "Miles"),
34);
c1m2a1i5.AjouteDomaine("pègre");
c1m2a1i5.AjouteDomaine("cambriolages");
c1m2a1.AjouteInformateur(c1m2a1i5);
c1.AjouteAgentMission("Clématite", c1m2a1);
Agent c1m3a0 = new Agent(new Identité("Addison", "Reynolds"), 43);
Informateur c1m3a0i0 = new Informateur(new Identité("Charlotte", "Hunt"),
40);
c1m3a0i0.AjouteDomaine("drogue");
c1m3a0.AjouteInformateur(c1m3a0i0);
Informateur c1m3a0i1 = new Informateur(new Identité("Aiden", "Bryan"),
20);
c1m3a0i1.AjouteDomaine("drogue");
c1m3a0i1.AjouteDomaine("racket");
c1m3a0i1.AjouteDomaine("mafia");
c1m3a0.AjouteInformateur(c1m3a0i1);
Informateur c1m3a0i2 = new Informateur(new Identité("Ella", "Fletcher"),
29);
c1m3a0i2.AjouteDomaine("cambriolages");
c1m3a0.AjouteInformateur(c1m3a0i2);
Informateur c1m3a0i3 = new Informateur(new Identité("Hailey",
"Fletcher"), 31);
c1m3a0i3.AjouteDomaine("pègre");
c1m3a0i3.AjouteDomaine("blanchiment d'argent");
c1m3a0.AjouteInformateur(c1m3a0i3);
Informateur c1m3a0i4 = new Informateur(new Identité("Scarlett", "Lee"),
37);
c1m3a0i4.AjouteDomaine("cambriolages");
c1m3a0.AjouteInformateur(c1m3a0i4);
Informateur c1m3a0i5 = new Informateur(new Identité("Emily", "Roberts"),
40);
c1m3a0i5.AjouteDomaine("contrebande");
c1m3a0i5.AjouteDomaine("blanchiment d'argent");
c1m3a0.AjouteInformateur(c1m3a0i5);
c1.AjouteAgentMission("Lys", c1m3a0);
Agent c1m3a1 = new Agent(new Identité("Hailey", "Graham"), 20);
Informateur c1m3a1i0 = new Informateur(new Identité("Jayce", "Nichols"),
30);
c1m3a1i0.AjouteDomaine("espionnage technique");
c1m3a1.AjouteInformateur(c1m3a1i0);
Informateur c1m3a1i1 = new Informateur(new Identité("Oliver", "Wood"),
20);
c1m3a1i1.AjouteDomaine("contrebande");
c1m3a1i1.AjouteDomaine("crime organisé");
c1m3a1.AjouteInformateur(c1m3a1i1);
c1.AjouteAgentMission("Lys", c1m3a1);
```

```
Agent c1m3a2 = new Agent(new Identité("Hailey", "Hill"), 39);
Informateur c1m3a2i0 = new Informateur(new Identité("Mackenzie", "Kerr"),
33);
c1m3a2i0.AjouteDomaine("crime organisé");
c1m3a2.AjouteInformateur(c1m3a2i0);
Informateur c1m3a2i1 = new Informateur(new Identité("Addison", "Knight"),
25);
c1m3a2i1.AjouteDomaine("mafia");
c1m3a2i1.AjouteDomaine("blanchiment d'argent");
c1m3a2i1.AjouteDomaine("pègre");
c1m3a2.AjouteInformateur(c1m3a2i1);
Informateur c1m3a2i2 = new Informateur(new Identité("Nicholas",
"Hunter"), 30);
c1m3a2i2.AjouteDomaine("mafia");
c1m3a2.AjouteInformateur(c1m3a2i2);
Informateur c1m3a2i3 = new Informateur(new Identité("Nicholas", "Davis"),
28);
c1m3a2i3.AjouteDomaine("blanchiment d'argent");
c1m3a2.AjouteInformateur(c1m3a2i3);
Informateur c1m3a2i4 = new Informateur(new Identité("Victoria", "Kerr"),
44);
c1m3a2i4.AjouteDomaine("mafia");
c1m3a2.AjouteInformateur(c1m3a2i4);
Informateur c1m3a2i5 = new Informateur(new Identité("Landon", "Ellis"),
22);
c1m3a2i5.AjouteDomaine("mafia");
c1m3a2i5.AjouteDomaine("pègre");
c1m3a2i5.AjouteDomaine("racket");
c1m3a2.AjouteInformateur(c1m3a2i5);
c1.AjouteAgentMission("Lys", c1m3a2);
Agent c1m4a0 = new Agent(new Identité("Dylan", "Saunders"), 37);
Informateur c1m4a0i0 = new Informateur(new Identité("Jayden", "Ellis"),
46);
c1m4a0i0.AjouteDomaine("racket");
c1m4a0i0.AjouteDomaine("blanchiment d'argent");
c1m4a0.AjouteInformateur(c1m4a0i0);
Informateur c1m4a0i1 = new Informateur(new Identité("Ethan", "Wheeler"),
45);
c1m4a0i1.AjouteDomaine("drogue");
c1m4a0i1.AjouteDomaine("espionnage technique");
c1m4a0i1.AjouteDomaine("vente d'armes");
c1m4a0.AjouteInformateur(c1m4a0i1);
c1.AjouteAgentMission("Joubarde", c1m4a0);
Agent c1m4a1 = new Agent(new Identité("Peyton", "Walker"), 24);
Informateur c1m4a1i0 = new Informateur(new Identité("Mackenzie",
"Coleman"), 40);
c1m4a1i0.AjouteDomaine("cambriolages");
```

```
c1m4a1i0.AjouteDomaine("vente d'armes");
c1m4a1i0.AjouteDomaine("racket");
c1m4a1.AjouteInformateur(c1m4a1i0);
Informateur c1m4a1i1 = new Informateur(new Identité("Nathan", "Dunn"),
47);
c1m4a1i1.AjouteDomaine("blanchiment d'argent");
c1m4a1i1.AjouteDomaine("vente d'armes");
c1m4a1.AjouteInformateur(c1m4a1i1);
Informateur c1m4a1i2 = new Informateur(new Identité("Camilla", "Adams"),
38);
c1m4a1i2.AjouteDomaine("mafia");
c1m4a1.AjouteInformateur(c1m4a1i2);
c1.AjouteAgentMission("Joubarde", c1m4a1);
c1.AjouteAgentMission("Joubarde", c1m0a1);
Agent c1m4a2 = new Agent(new Identité("Mason", "Brown"), 46);
Informateur c1m4a2i0 = new Informateur(new Identité("Mackenzie",
"Henry"), 24);
c1m4a2i0.AjouteDomaine("vente d'armes");
c1m4a2i0.AjouteDomaine("cambriolages");
c1m4a2.AjouteInformateur(c1m4a2i0);
Informateur c1m4a2i1 = new Informateur(new Identité("Lucas", "Barrett"),
30);
c1m4a2i1.AjouteDomaine("espionnage technique");
c1m4a2.AjouteInformateur(c1m4a2i1);
Informateur c1m4a2i2 = new Informateur(new Identité("Caleb", "Sanders"),
27);
c1m4a2i2.AjouteDomaine("cambriolages");
c1m4a2.AjouteInformateur(c1m4a2i2);
c1.AjouteAgentMission("Joubarde", c1m4a2);
```