TP de C# .NET

OBJECTIFS

Ce document est une aide pour la réalisation des travaux pratiques de C#. Il vous propose notamment différents diagrammes comme support. Il ne se substitue pas entièrement aux explications données en cours ou en TP.

TABLE DES MODIFICATIONS

Auteur	Modifications	Version	Date
Marc Chevaldonné	Description du TP 1 + exercices de compléments	1.0	05 septembre 2013
Marc Chevaldonné	Description du TP 2 + exercices de compléments	2.0	14 septembre 2013

TP 1

Les objectifs de ce premier TP sont :

- la découverte de Visual Studio 2012
- la découverte de la structure des projets (solutions, projets, fichiers de code...)
- l'écriture d'une classe (membres, méthodes)
- l'utilisation de types (int, float, bool, string, DateTime)
- l'écriture de types personnalisés (enum, classe, classe de test)
- la création d'une bibliothèque de classes et d'une application Console
- l'utilisation du framework .NET (Random, Globalization)
- ...

PRÉPARATION DU PROJET ET DE LA SOLUTION

Créez une solution vide.

Ajoutez-lui une bibliothèque de classes.

ÉCRITURE D'UNE CLASSE

Ajoutez une classe Etudiant.

Ajoutez-lui deux membres pour le nom et le prénom de l'étudiant.

Ajoutez des accesseurs pour lire publiquement et écrire en privé ces deux membres.

N'oubliez pas de commenter.

Ajoutez un constructeur pour initialiser des instances d'Etudiant.

TEST

Ajoutez un projet de type Application Console à la solution.

Ajoutez-lui la référence vers votre bibliothèque de classes précédente.

Testez la construction et l'affichage des nom et prénom d'un étudiant.

Testez en débogage et sans débogage.

DEBUG

Dans la bibliothèque de classes, utilisez la classe Debug (System.Diagnostics) pour faire des affichages dans la fenêtre de sortie. Testez le résultat en Debug et en Release.

RÉSULTATS

Observez les résultats produits par les deux projets et modifiez les chemins de sortie pour les résultats soient tous dirigés vers le même dossier.

MANIPULATION DE STRING

Ecrivez une méthode privée qui met des majuscules devant chaque nom propre (exemple : «jean-louis» devient «Jean-Louis» et «imbert» devient «Imbert»). Utilisez cette méthode dans les setters de nom et prénom.

TOSTRING

Affichez un étudiant sous la forme : «Prénom Nom» à l'aide de ToString (exemple : «Jean-Louis Imbert»). Testez ToString dans l'application Console.

DATETIME

Ajoutez un membre DateDeNaissance à votre classe. Ajoutez les accesseurs et modifiez le constructeur. Mettez à jour ToString pour qu'il affiche l'étudiant sous la forme suivante : «Prénom Nom (DateDeNaissance)» (par exemple : «Jean-Louis Imbert (15/09/1968)». Mettez le test à jour.

ENUMÉRATION

Créez un nouveau type permettant de proposer à travers une énumération les différents sexes possibles (Inconnu, Homme, Femme). Ajoutez à la classe Etudiant un membre de ce nouveau type. Modifiez le constructeur et la méthode ToString afin d'obtenir «Prénom Nom (S, DateDeNaissance)» où S vaut H pour homme, F pour femme et rien pour Inconnu. Mettez le test à jour.

TABLEAU

Ajoutez un tableau de 10 notes à votre classe Etudiant. Initialisez le tableau de notes à 10 notes valant -1. Ajoutez une méthode GetNotes rendant une copie de ce tableau.

MÉTHODES

Ajoutez deux membres à votre classe donnant la note minimale et la note maximale de cet Etudiant. Ajoutez une méthode GénérerNote() privée qui rendra un nombre aléatoire compris entre les deux membres précédents. Ajoutez une méthode ExamenBlanc() qui rendra une note obtenue grâce à la méthode précédente. Ajoutez une méthode Examen qui générera

une note grâce à GénérerNote(), la rentrera dans la première case du tableau dont la valeur est égale à -1. Si une telle case existe, la méthode rendra true ainsi que la note dans un paramètre passé par référence. Si toutes les cases sont occupées, la méthode rend false.
Ajoutez une méthode Travaille augmentant de 1 la note minimale et de 2 la note maximale. Faites en sorte que ExamenBlanc appelle Travaille. (Attention de bien borner note minimale à 20 et note maximale à 21). Ajoutez une méthode qui calcule la moyenne.
Mettez le test à jour pour appeler plusieurs fois Travaille, ExamenBlanc, Examen...

NULLABLETYPE

Plutôt que d'utiliser -1 comme «non note», nous allons préférer l'utilisation d'un NullableType. Modifiez votre tableau de notes pour qu'ils ne contiennent plus des entiers, mais des int?. Modifiez votre classe en conséquence. Testez-la.

Compléments pour le cours 1

En complément du TP1, voici d'autres exercices sur des thèmes abordés lors du premier cours, que je vous conseille de faire afin de vous entrainer davantage.

EXERCICE 01_01

tableaux

Soit un tableau T ayant N éléments entiers quelconques où N est une constante entière. Écrire un programme en C# permettant de :

- faire la saisie du tableau,
- calculer la somme de ses éléments.

EXERCICE 01_02

tableaux à deux dimensions

Soit un tableau T à 5 lignes et 6 colonnes. Chaque élément est un entier. Écrire un programme en C# permettant de :

- faire la saisie du tableau,
- calculer la somme de ses éléments.

EXERCICE 01_03

tableaux à deux dimensions

Créer dans un tableau P à 2 dimensions les 10 premières lignes du triangle de Pascal. Chaque élément du triangle de Pascal est obtenu par la formule :

$$P[L, C] = P[L-1, C-1] + P[L-1, C] \qquad \text{avec } L, C > 1$$

La première colonne est initialisée à 1, le reste du tableau à 0

Résultat :

1	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0
1	3	3	1	0	0	0	0	0	0
1	4	6	4	1	0	0	0	0	0
1	5	10	10	5	1	0	0	0	0
1	6	15	20	15	6	1	0	0	0
1	7	21	35	35	21	7	1	0	0
1	8	28	56	70	56	28	8	1	0
1	9	36	84	126	126	84	36	9	1

On affichera uniquement le triangle de Pascal à la fin du programme.

EXERCICE 01_04

valeurs et références

- Faire une méthode statique (et l'utiliser) prenant deux paramètres par valeur et retournant un réel égal à la somme des deux.
- Faire une méthode statique (et l'utiliser) ne rendant rien et prenant deux paramètres par valeur et un troisième paramètre par référence qui devra être égal à la fin de la méthode à la somme des deux premiers. Faites cet exercice une fois avec `ref` et une autre fois avec `out`.
- Faire une méthode statique (et l'utiliser) prenant un tableau d'entiers en paramètre et rendant la somme des éléments du tableau.
- Faire une méthode statique (et l'utiliser) prenant un nombre indéterminé d'entiers en paramètres et rendant la somme des ces entiers.

EXERCICE 01_05

passage de paramètres par référence

On considère une suite (un) définie par :

$u_n = u_{n-1} + u_{n-2} \quad (n \geq 3)$

$u_1 = 1$

$u_2 = 2$

appelée suite de Fibonacci.

- Écrire le programme permettant de déterminer la valeur et le rang du premier terme de cette suite supérieur ou égal à 10000, à l'aide d'une boucle `while`.

- Recommencer en utilisant une méthode et des passages de paramètres par référence, pour remplacer les instructions de chaque itération de la boucle `while`, et mettre à jour toutes les variables nécessaires.

EXERCICE 01_06

foreach

Calculez le maximum de trois nombres entiers entrés au clavier à l'aide d'un `foreach`. Pour cela, utilisez un tableau pour stocker les nombres entrés au clavier.

EXERCICE 01_07

foreach

Écrivez le programme qui calcule la somme des N premiers nombres entiers positifs à l'aide d'un `foreach`. Utilisez un tableau.

EXERCICE 01_08

for vs foreach

Soit un tableau d'entiers. À l'aide d'une boucle `for`, incrémentez chaque entier du tableau de 1. Essayez de réaliser la même opération avec une boucle `foreach`. Que constatez-vous ?

EXERCICE 01_09

classe et méthodes
surcharge de méthodes

Écrivez une classe `Voiture`. Les voitures de cette classe possèdent un régulateur de vitesse. Ajoutez à cette classe `Voiture` :

- un attribut `mVitesse` (entier compris entre 0 et 130 inclus),
- un *getter* rendant la vitesse de la voiture,
- un *setter* privé permettant de fixer une vitesse donnée et vérifiant que la vitesse rentrée est valide,
- une méthode permettant d'incrémenter la vitesse de 2 km/h
- et une méthode permettant de décrémenter la vitesse de 2 km/h.

Surchargez ensuite la méthode d'incrémentation en lui passant le nombre de vitesses à incrémenter.

Avant de coder cette classe, réalisez un diagramme de classes UML au brouillon. Réalisez ensuite un projet de test pour tester les méthodes.

EXERCICE 01_10

Math
Random

Écrivez un programme qui affiche à l'écran un nombre entier x choisi aléatoirement entre `-100` et le nombre entier le plus grand autorisé pour un `int` et son image f(x) réelle ou f est la fonction continue par morceaux suivante :

$f(x) = 2x + 5e^x$ si $x < -1$

$f(x) = 3x^2 - 4\sqrt{(|5x-1|)}$ si $x >= -1$

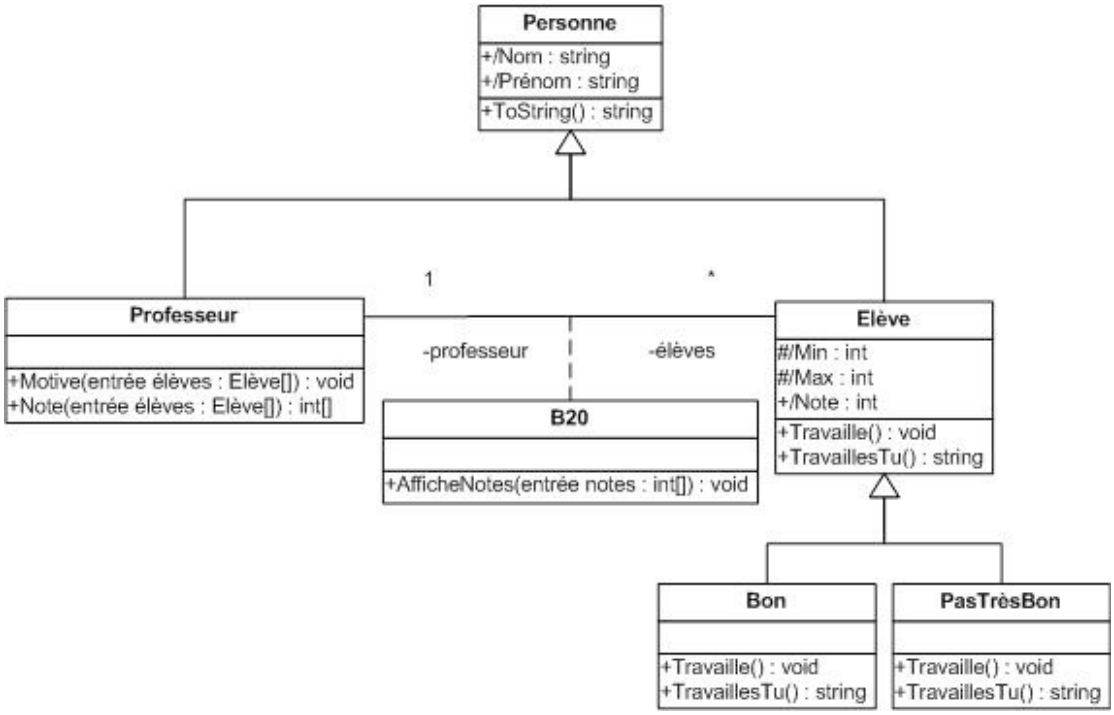
TP 2

Les objectifs de ce deuxième TP sont :

- l'écriture de classes et de propriétés,
- l'utilisation de l'héritage,
- l'utilisation du polymorphisme,
- l'écriture et l'implémentation d'interfaces.

HÉRITAGE, RÉÉCRITURE D'UNE MÉTHODE (NEW), TYPAGE STATIQUE

On souhaite réaliser un programme qui simule une relation professeur-élèves dans la salle B20, comme proposé dans le diagramme de classes partiel ci-dessous.



Une `Personne` possède un `Nom` et un `Prénom`.

Les classes `Professeur` et `Elève` sont des spécialisations de la classe `Personne`.

Un `Elève` possède deux propriétés `Min` et `Max` représentant respectivement la note minimale et la note maximale qu'il obtiendra lors de la notation.

La propriété calculée `Note` de `Elève` choisit un nombre entier aléatoire entre `Min` et `Max`.

Un `Elève` peut travailler, ce qui augmente ses propriétés `Min` et `Max`.

Un `Elève` peut répondre à la question «travailles-tu ?» en renvoyant un message sous la forme d'une chaîne de caractères.

Un `Professeur` peut motiver des `Elèves`. L'appel de la méthode `Motive(Elève[])` appelle la méthode `Travaille()` des `Elèves` passés en paramètres.
Un `Professeur` peut noter des `Elèves`. L'appel de la méthode `Note(Elève[])` récupère les notes des `Elèves` passés en paramètres (via leur propriété calculée `Note`).
Les différents élèves sont généralisés par la classe `Elève` et celle-ci est spécialisée dans les sous-classes `Bon` et `PasTrèsBon` qui réécrivent les méthodes Travaille et TravaillesTu ; leurs différences sont présentées dans le tableau suivant :

	Elève	Bon	PasTrèsBon
Min initial	0	8	0
Max initial	0	12	4
Travaille	augmente Min de 1 augmente Max de 2	Travaille deux fois plus qu'un Elève	augmente Min de 1 si Min +Max est pair augmente Max de 1 si Min +Max est impair
TravaillesTu	«*Un peu*» si la note est inférieure à 5 «*Beaucoup*» si la note est inférieure à 10 «*Passionnément*» si la note est inférieure à 15 «*À la folie*» pour les autres notes	«*Oui*»	«*Oui, mais j'ai du mal*»

Pour toutes les classes `Elève`, le retour de la méthode `TravaillesTu` est suivi de «*[Min, Max]*» où *Min* et *Max* sont remplacés respectivement par les valeurs des propriétés `Min` et `Max` (pour permettre de vérifier les résultats dans l'affichage Console).
La classe `B20`, qui remplace la classe `Program` et possède donc la méthode statique `Main`, créera une instance de `Professeur` et un tableau de 6 `Elèves` en utilisant les données du tableau suivant :

Type	Prénom	Nom
Professeur	*Roger*	*Rabbit*
Elève	*Schtroumpf*	*Normal*
Elève	*Schtroumpf*	*Normal 2*
Bon	*Schtroumpf*	*Bon*
Bon	*Schtroumpf*	*Bon 2*
PasTrèsBon	*Schtroumpf*	*PasTrèsBon*
PasTrèsBon	*Schtroumpf*	*PasTrèsBon 2*

La classe `B20` possède également une méthode `AfficheNotes` qui permet l'affichage des notes attribuées dans une interface de type Console. Cette méthode affichera :

```
Notes du Professeur Prénom Nom :
Prénom Nom : Note/20 ; travailles-tu ? réponse
Prénom Nom : Note/20 ; travailles-tu ? réponse
Prénom Nom : Note/20 ; travailles-tu ? réponse
```

La méthode `Main` réalisera ensuite les opérations suivantes :

- Roger Rabbit note ses élèves,
- B20 affiche les notes,
- Roger Rabbit motive 5 fois ses élèves,
- Roger Rabbit note ses élèves,
- B20 affiche les notes,
- Roger Rabbit motive 5 fois ses élèves,
- Roger Rabbit note ses élèves,
- B20 affiche les notes,
- Roger Rabbit motive 5 fois ses élèves,
- Roger Rabbit note ses élèves,
- B20 affiche les notes.

Lancez plusieurs fois le programme et observez les résultats. Remarquez notamment que le tableau d'`Elèves` est de type `Elève[]`, alors que les six instances ont été construites avec des types différents : `Elève`, `Bon`, `PasTrèsBon`. Toutefois, les méthodes `Travaille` et `TravaillesTu` appelées sont toujours celles de `Elève`. Il s'agit bien d'un typage statique.

HÉRITAGE, MÉTHODES VIRTUELLES, POLYMORPHISME, TYPAGE DYNAMIQUE

Reprenez l'exercice précédent et rendez les méthodes `Travaille` et `TravaillesTu` dans la classe `Elève`, virtuelles. Effectuez les modifications qui s'imposent et relancez le programme. Que constatez-vous ?

Remarquez que le tableau d'`Elèves` est toujours de type `Elève[]` et les instances sont toujours construites avec des types différents : `Elève`, `Bon`, `PasTrèsBon`. Mais désormais, les méthodes `Travaille` et `TravaillesTu` dépendent du type utilisé à la construction (type dynamique) et pas celui déclaré (type statique, ici `Elève`). `Professeur` appelle la méthode `Travaille` d'un `Elève`, sans connaître son type, et pourtant, il appelle la méthode `Travaille` de ce type concret qui lui est inconnu. Vous venez de mettre en pratique le polymorphisme et le typage dynamique.

HÉRITAGE, MÉTHODES VIRTUELLES, MÉTHODES ABSTRAITES, CLASSES ABSTRAITES

Reprenez l'exercice précédent et rendez la méthode `Travaille` dans la classe `Elève`, abstraite. Effectuez les modifications qui s'imposent. Ajoutez une nouvelle classe `Fainéant`, sous-classe de `Elève`, avec les caractéristiques suivantes :

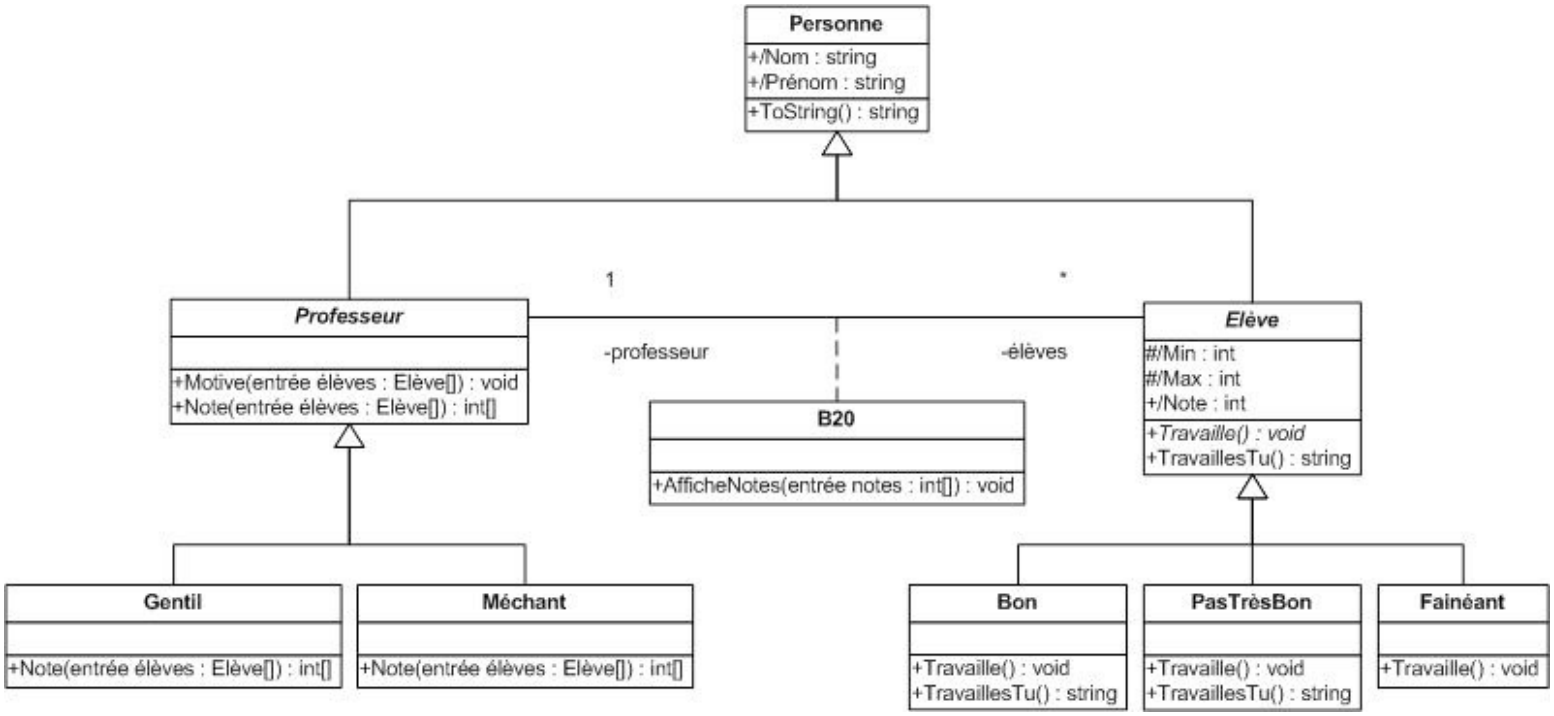
	`Fainéant`
`Min` initial	`0`
`Max` initial	`0`
`Travaille`	a 1 chance sur 3 de : • augmenter `Min` de 1 • et augmenter `Max` de 2
`TravaillesTu`	même réponse qu'`Elève`

Modifiez le tableau d'`Elèves` (toujours de type `Elève[]`) pour avoir une instance de chaque type concret.

Rendez maintenant la méthode `Note` dans la classe `Professeur`, virtuelle et la classe `Professeur` abstraite. Ajoutez deux nouvelles classes, sous-classes de `Professeur` : `Gentil` et `Méchant`.

Type	Note
`Gentil`	Note les élèves comme `Professeur` puis rajoute un point à chaque `Elève` ayant une note strictement inférieure à 20.
`Méchant`	Note les élèves comme `Professeur` puis enlève un point à chaque `Elève` ayant une note strictement supérieure à 0.

Le diagramme de classes ci-dessous résume toutes les modifications.



Modifiez ensuite la méthode `B20.Main` pour que :

- un `Professeur Gentil` «*`Roger Rabbit`*» note les élèves,
- `B20` affiche les notes,
- *`Roger Rabbit`* motive 5 fois les élèves,
- *`Roger Rabbit`* note les élèves,
- `B20` affiche les notes,
- *`Roger Rabbit`* motive 5 fois les élèves,
- *`Roger Rabbit`* note les élèves,
- `B20` affiche les notes,
- *`Roger Rabbit`* tombe malade et est remplacé par un `Professeur Méchant` «*`Harvey Dent`*»,
- *`Harvey Dent`* motive 5 fois les élèves,

• *Harvey Dent* note les élèves,

• B20 affiche les notes.

Constatez que si une classe possède une méthode abstraite, alors la classe doit être déclarée comme abstraite et ne peut pas être instanciée.

Constatez que si une classe est déclarée comme abstraite, elle ne peut pas être instanciée mais ne possède pas nécessairement de méthode abstraite.

Dans une classe abstraite, on pourra donc utiliser des méthodes virtuelles pour faire des implémentations par défaut (cf. `TravaillesTu` dans `Elève` : `Fainéant` n'a pas besoin de la réécrire) et des méthodes abstraites pour forcer leur réécriture (cf. `Travaille` dans `Elève` et dans toutes ses sous-classes).

INTERFACES, IMPLÉMENTATION D'UNE INTERFACE, IMPLÉMENTATION MULTIPLE D'INTERFACES

Reprenez l'exercice précédent et faites les modifications résumées dans le diagramme de classes ci-dessous et les explications suivantes.

1. Ajoutez une interface `ITricheur` possédant une méthode `Triche`.

Créez une nouvelle sous-classe `Tricheur` de `Elève`, implémentant l'interface `ITricheur` et réalisant les modifications suivantes :

	Tricheur
Travaille	vide
TravaillesTu	même réponse qu'`Elève`
Triche	passe `Min` à 15 passe `Max` à 15

Relancez le programme et vérifiez que `Tricheur` s'en sort bien...

2. Ajoutez une interface `ICafé` possédant une méthode `PrépareLeCafé` (le retour de cette méthode sera un message de gentillesse accompagnant le café).

Modifiez la classe `Professeur` en lui ajoutant une méthode `DemandeUnCafé` prenant un paramètre de type `ICafé`. Cette méthode (non virtuelle) rendra une chaîne de caractères sur deux lignes contenant sur la première ligne «*Prénom Nom* `demande un café`» et sur la deuxième ligne, le retour de la méthode `PrépareLeCafé` appelée sur le paramètre de type `ICafé` passé en paramètre.

Créez une nouvelle sous-classe `Fayot` de `Bon` implémentant l'interface `ICafé` et réalisant les modifications suivantes :

Personne
+/Nom : string
+/Prénom : string
+ToString() : string
Professeur
+Motive(entrée élèves : Elève[]) : void
+Note(entrée élèves : Elève[]) : int[]
+DemandeUnCafé(entrée serveur : ICafé) : string
Gentil
+Note(entrée élèves : Elève[]) : int[]
+PrépareLeCafé() : string
Méchant
+Note(entrée élèves : Elève[]) : int[]
ICafé
B20
+AfficheNotes(entrée notes : int[]) : void
-professeur
1
-élèves
*
Elève
#/Min : int
#/Max : int
+/Note : int
+Travaille() : void
+TravaillesTu() : string
Bon
+Travaille() : void
+TravaillesTu() : string
PasTrèsBon
+Travaille() : void
+TravaillesTu() : string
Fainéant
+Travaille() : void
Tricheur
+Travaille() : void
+Triche() : void
Fourbe
+Travaille() : void
+TravaillesTu() : string
+Triche() : void
+PrépareLeCafé() : string
Fayot
+PrépareLeCafé() : string
ICafé
«interface»
ITricheur
+Triche() : void
«interface»
ICafé
+PrépareLeCafé() : string

	Fayot
`Travaille`	comme `Bon`
`TravaillesTu`	«Bien sûr !»
`PrépareLeCafé`	«Voici un bon café ! Et n'oubliez pas que c'est de la part de *Prénom Nom*»

Modifiez la classe `Gentil` pour qu'elle implémente `ICafé`. Le retour de `PrépareLeCafé` sera : «`Voici un café de la part de ton collègue` *`Prénom Nom`*».

Modifiez `B20.Main` pour que `Harvey Dent` demande un café à `Schtroumpf Fayot`, puis demande un café à `Roger Rabbit`.

3. Créez une nouvelle sous-classe `Fourbe` de `Elève`, implémentant `ITricheur` et `ICafé`, en utilisant les modifications suivantes :

	Fourbe
`Travaille`	augmente `Min` de 1 et `Max` de 1
`TravaillesTu`	«Avec plaisir !»
`PrépareLeCafé`	Augmente `Max` de 2 et rend : «Voici un bon café ! Et n'oubliez pas que c'est de la part de *Prénom Nom*»
`Triche`	double `Min` et double `Max`

Modifiez `B20.Main` pour que `Schtroumpf Fourbe` triche et que `Roger Rabbit` lui demande un café.

Compléments pour le cours 2

En complément du TP2, voici d'autres exercices sur des thèmes abordés lors des 2 premiers cours, que je vous conseille de faire afin de vous entrainer davantage.

EXERCICE 02_01

propriétés

Modifiez la classe `Voiture` de l'exercice 01_09 en utilisant une propriété à la place du *getter* public et du *setter* privé pour lire et écrire la vitesse.

EXERCICE 02_02

propriétés automatiques et calculées

Écrivez une classe `Vecteur`, contenant trois réels `X`, `Y`, `Z`. N'écrivez pas de membres et utilisez uniquement des propriétés automatiques. Ajoutez une propriété calculée `Norme` en lecture seule.

EXERCICE 02_03

propriétés automatiques et calculées

Écrivez une classe `Pavé`, contenant trois réels `Largeur`, `Longueur`, `Hauteur`. N'écrivez pas de membres et utilisez uniquement des propriétés automatiques. Ajoutez une propriété calculée `Volume` en lecture seule.

EXERCICE 02_04

ToString

Reprenez la classe `Vecteur` de l'exercice 02_02 et réécrivez la méthode `ToString` pour qu'elle affiche :

`{ X, Y, Z }` où `X`, `Y`, `Z` sont remplacés par les valeurs numériques

EXERCICE 02_05

ToString

Reprenez la classe `Pavé` de l'exercice 02_03 et réécrivez la méthode

`ToString` pour
qu'elle affiche :

`L = L, l = L, h = h`, où

`L, L, h` seront remplacés par les longueur, largeur, hauteur.

EXERCICE 02_06

string.Format

Reprenez la classe `Vecteur` de l'exercice 02_04 et réécrivez la méthode `ToString` en utilisant un `string.Format` pour qu'elle affiche : `{ X, Y, Z }` où `X, Y, Z` sont remplacés par les valeurs numériques.

EXERCICE 02_07

StringBuilder

Reprenez la classe `Pavé` de l'exercice 02_05 et réécrivez la méthode `ToString` à l'aide d'un `StringBuilder` pour qu'elle affiche : `L = L, l = L, h = h`, où `L, L, h` seront remplacés par les longueur, largeur, hauteur.

EXERCICE 02_08

struct

Reprenez la classe `Vecteur` de l'exercice 02_06 (ou 02_04 ou 02_02) et faites les modifications qui s'imposent (aidez-vous du compilateur) pour en faire une structure.

EXERCICE 02_09

struct

Reprenez la classe `Pavé` de l'exercice 02_07 (ou 02_05 ou 02_03) et faites les modifications qui s'imposent (aidez-vous du compilateur) pour en faire une structure.

EXERCICE 02_10

value vs reference

Reprenez la structure `Pavé` de l'exercice 02_09. Modifiez les propriétés de ses dimensions pour permettre leur modification par l'utilisateur de la structure (*ie* modifiez la visibilité des setters).

Écrivez un programme de test qui réalise les opérations suivantes :

- construction de `pavé1` de type `Pavé`, avec `L=30`, `l=20` et `h=10`,
- construction de `pavé2` de type `Pavé`, avec `L=25`, `l=15` et `h=5`,
- affichage du `ToString` de `pavé1` et `pavé2`,

- `pavé2 = pavé1`,
- affichage du `ToString` de `pavé1` et `pavé2`,
- modification des dimensions de `pavé2`,
- affichage du `ToString` de `pavé1` et `pavé2`.

Transformez la structure `Pavé` en classe (changez uniquement le mot clé `struct` en `class`) et observez les différences à l'exécution. Expliquez-les.

EXERCICE 02_11

indexeurs

Soit la classe `ClassementCurling` suivante. Cette classe contient deux tableaux de même taille contenant (tableau 1) les noms des pays dans l'ordre du classement de curling et (tableau 2) le points de ces pays.

```
public class ClassementCurling
{
   string[] mPays = new string[] {"Canada",
              "Scotland/Great Britain", "Norway", "Sweden",
              "Switzerland", "Germany", "USA", "France",
       "China", "Denmark", "Czech Republic",
              "New Zealand", "Italy", "Korea", "Finland",
              "Australia", "Russia", "Ireland", "Latvia"} ;

   int[] mPoints = new int[] {1064, 784, 774, 619, 550, 451,
              434, 426, 402, 384, 192, 179, 141, 128, 122,
       120, 115, 107, 105} ;
}
```

a) Ajoutez à cette classe un indexeur permettant de rendre le `i`ème pays dans le classement. Par exemple si `cc` est une instance de `ClassementCurling`, `cc[8]` vaut `China`. Cet indexeur vérifiera que l'index passé entre crochets est bien dans les limites du tableau, sinon il rendra `null`. Testez votre indexeur dans un projet de test.

b) Ajoutez à cette classe un indexeur permettant de rendre le nombre de points du pays dont le nom est passé entre crochets. Par exemple si `cc` est une instance de `ClassementCurling`, `cc[«Russia»]` rend `115`. Cet indexeur vérifiera que la chaîne de caractères passée entre crochets est bien dans le tableau, sinon il rendra `-1`. Testez votre indexeur dans un projet de test.

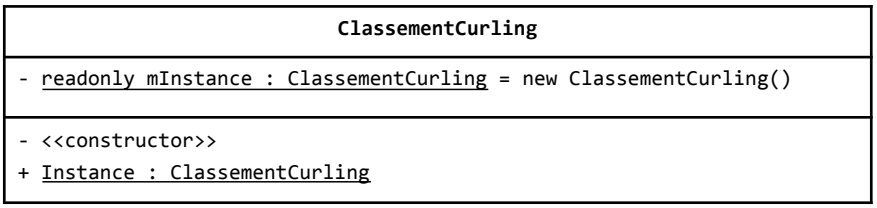
EXERCICE 02_12

static

Soit la classe `ClassementCurling` de l'exercice précédent. On veut s'assurer qu'une seule instance de cette classe peut exister. Modifiez la classe `ClassementCurling` en implémentant le *design pattern* singleton.

Note : Il n'est pas nécessaire d'avoir fait l'exercice 02_11 pour réaliser cet exercice.

Rappel sur le singleton avec un diagramme de classes UML (vous pourrez par exemple utiliser une propriété pour `Instance`*, et l'initialiseur pour construire le membre statique* `mInstance`*) :*



`Instance` *rend l'instance unique de* `ClassementCurling`.

EXERCICE 02_13

write-once immutability

Soit la classe `Cible` suivante :

```
public class Cible
{
   public string Description
   {
      get;
      set;
   }

   public int Hauteur
   {
      get;
      private set;
   }

   public int Largeur
   {
      get;
      private set;
   }

   public void ChangeDimensions(int hauteur, int largeur)
   {
      Hauteur = hauteur;
      Largeur = largeur;
```

```
    }

    public Cible (string desc, int hauteur, int largeur)
    {
        Description = desc;
        Hauteur = hauteur;
        Largeur = largeur;
    }

    public override string ToString ()
    {
        return string.Format ("[Cible: Description={0},
                               Hauteur={1}, Largeur={2}]",
                               Description, Hauteur, Largeur);
    }
}
```

Soit la méthode `Main` de la classe `Program` suivante, qui utilise la classe `Cible` :

```
public static void Main (string[] args)
{
    Cible c = new Cible("Mickey", 100, 50);
    Console.WriteLine(c);
    c.Description = "King Kong";
    c.ChangeDimensions(200, 90);
    Console.WriteLine(c);
}
```

Faites de la classe `Cible` une structure immuable (*write-once immutable*) en faisant les modifications/suppressions qui s'imposent.

EXERCICE 02_14

write-once immutability

Quelques questions tordues pour bien comprendre...

Soit la classe `Point` et la structure `Cercle` suivantes :

```
public class Point
{
    public float X
    {
        get { return mX; }
    }
    private float mX;

    public float Y
    {
        get { return mY; }
    }
    private float mY;

    public Point(float x, float y)
    {
        mX = x;
        mY = y;
    }

    public void Translate(float tx, float ty)
    {
        mX += tx;
```

```
            mY += ty;
        }
    }

    public struct Cercle
    {
        public float Rayon
        {
            get { return mRayon; }
        }
        private readonly float mRayon;

        public Point Centre
        {
            get { return mCentre; }
        }
        private readonly Point mCentre;

        public Cercle(float rayon, Point centre)
        {
            mRayon = rayon;
            mCentre = centre;
        }
    }
```

La classe `Point` est-elle immuable ? La structure `Cercle` est-elle immuable ? Justifiez. Changez la classe `Point` en une structure (en changeant simplement `class` en `struct`). `Point` est-elle immuable ? `Cercle` est-elle immuable ? Justifiez.

EXERCICE 02_15

write-once immutability

La structure `Point` suivante n'est pas immuable, à cause de sa méthode `Translate`. Pourquoi ?

```
public struct Point
{
    public float X
    {
        get
        {
            return mX;
        }
    }
    private float mX;

    public float Y
    {
        get
        {
            return mY;
        }
    }
    private float mY;
```

```
   public Point(float x, float y)
   {
      mX = x;
      mY = y;
   }

   public void Translate(float tx, float ty)
   {
      mX += tx;
      mY += ty;
   }
}
```

Rendez cette structure immuable (*write-once*) tout en conservant la méthode `Translate`. Que faut-il changer dans cette méthode pour que la structure soit immuable ?

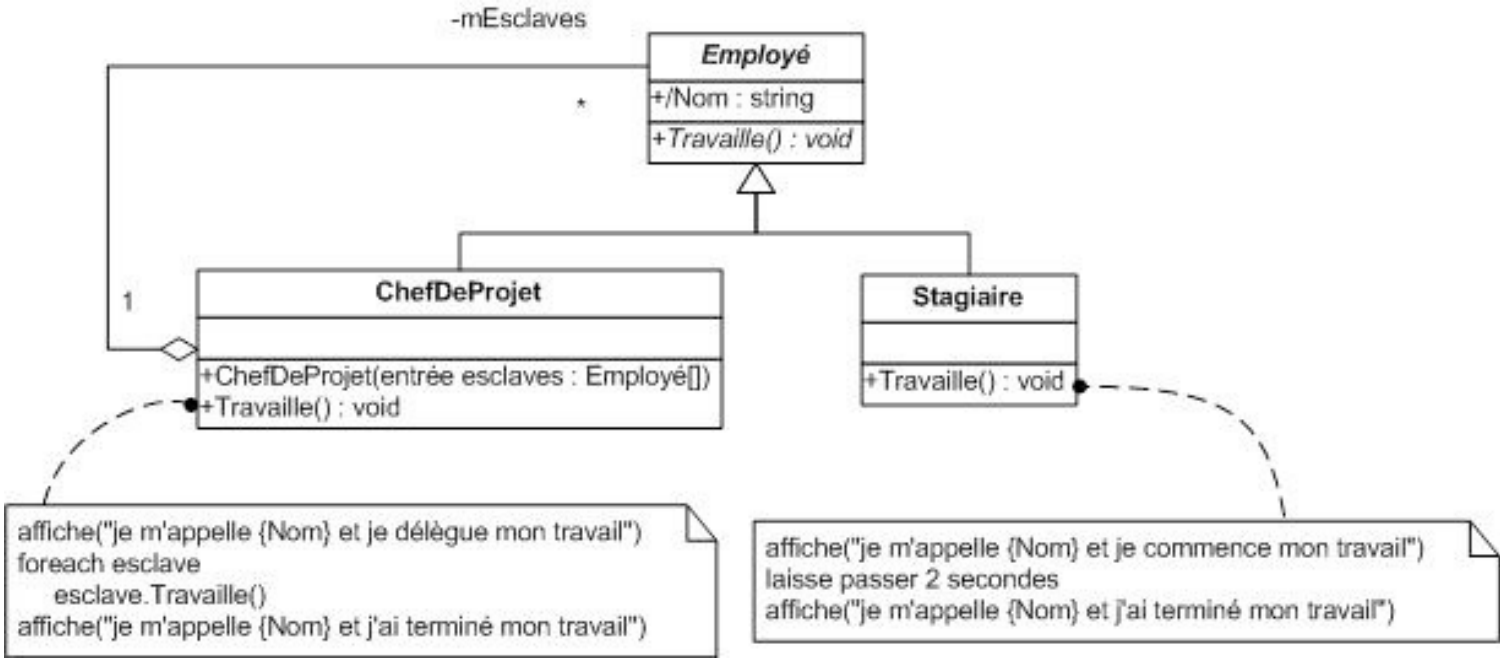
EXERCICE 02_16

NullableType

Reprenez la structure `Cible` issue de l'exercice 02_13. Soit la classe `TirÀLaCarabine` présentée plus bas. Cette classe représente un jeu de 5 cibles à abattre avec une carabine, en pleine fête foraine. L'indexeur permet de modifier les cibles lorsqu'elles sont touchées. Malheureusement, `Cible` étant une structure, et donc un type valeur, la valeur par défaut est une `Cible` sans `Description`, avec une `Hauteur` et une `Largeur` à `0`. Nous aurions préféré une `Cible` `null`. Ceci n'est possible qu'avec les types référence (classes). On peut toutefois le rendre possible avec un type valeur en utilisant un `NullableType`. Modifiez la classe `TirÀLaCarabine` pour qu'elle contienne 5 éléments de type «`Cible` ou `null`» et testez-la. Pour cela, utilisez la classe `Program` ci-dessous et modifiez le `ToString` de `TirÀLaCarabine` pour qu'il affiche `null` lorsqu'il n'y a pas de cible, ou la cible s'il y en a une.

```
public class TirÀLaCarabine
{
   const int NOMBRE_CIBLES = 5;

   Cible[] mCibles = new Cible[NOMBRE_CIBLES];

   public int NombreCibles
   {
      get { return mCibles.Length; }
   }

   public Cible this[int index]
   {
      get
```

```
        {
            if(index<0 || index >= mCibles.Length)
            {
                throw new IndexOutOfRangeException();
            }
            return mCibles[index];
        }
    }

    public TirÀLaCarabine(params Cible[] cibles)
    {
        int nbreMaxDeCibles = Math.Min(cibles.Length,
                                       mCibles.Length);
        for (int index = 0; index < nbreMaxDeCibles; index++)
        {
            mCibles[index] = cibles[index];
        }
    }

    public override string ToString ()
    {
        System.Text.StringBuilder sb =
                                new System.Text.StringBuilder();
        sb.AppendLine("Cibles :");
        foreach (Cible c in mCibles)
        {
```




```
            sb.AppendLine(c.ToString());
        }
        sb.AppendLine("**********************************");
        return sb.ToString();
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
                    Cible("Mickey", 90, 50),
TirÀLaCarabine                      new Cible("Gargamel", 150, 20),
tir = new                           new Cible("Schtroumpf à lunettes", 10, 5));
TirÀLaCarabine          Console.WriteLine(tir);
(                     }
                    }
new
```

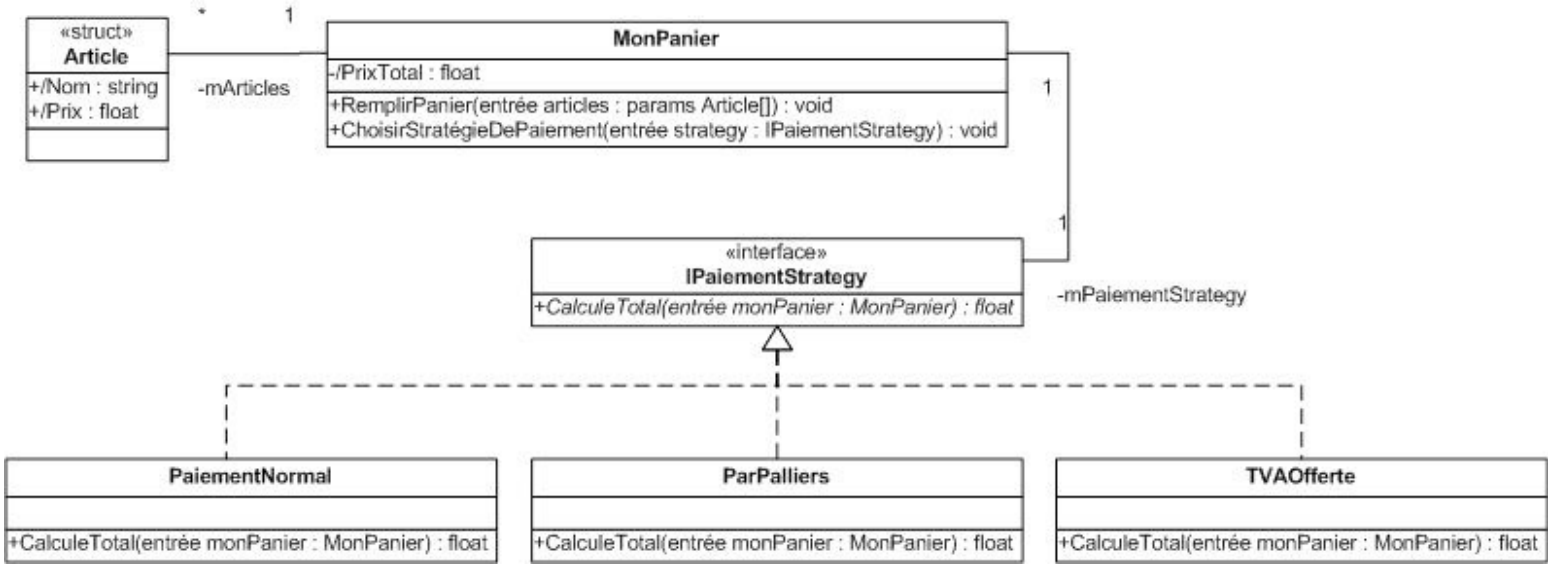
EXERCICE 02_17

héritage
polymorphisme
design pattern
Composite

Utilisez le *design pattern Composite* pour implémenter une hiérarchie basique au sein d'une entreprise comme proposée dans le diagramme de classes partiel ci-dessous : un `Employé` peut être `ChefDeProjet` ou `Stagiaire`.

Un `Employé` possède un `Nom` et une méthode abstraite `Travaille`.

Un `Stagiaire`, lorsqu'il travaille, réalise les trois opérations suivantes :



- affiche dans une interface de type Console : «`Je m'appelle Nom et je commence à travailler`»
- attend 2 secondes (`System.Threading.Thread.Sleep(2000);`)
- affiche dans une interface de type Console : «`Je m'appelle Nom et j'ai fini de travailler`».

Un `ChefDeProjet` possède des esclaves de type `Employé` (qui lui sont attribués à la construction de l'objet).

Un `ChefDeProjet`, lorsqu'il travaille, réalise les trois opérations suivantes :

- affiche dans une interface de type Console : «`Je m'appelle Nom et je délègue le travail à mes esclaves`»
- fait travailler ses esclaves
- affiche dans une interface de type Console : «`Je m'appelle Nom et j'ai (enfin, mes esclaves...) terminé mon travail`».

Pour tester vos classes, vous pouvez :

- créer dans la classe `Program`, un tableau de 8 `Employés` :
  - 4 `Stagiaires` : *`Kyle`*, *`Kévin`*, *`Ken`* et *`Karl`*,
  - 4 `ChefDeProjets` : *`Jim`* (chef de *`Kyle`* et *`Kévin`*), *`John`* (chef de *`Kévin`* et *`Jim`*), *`James`* (chef de *`Kyle`*, *`Karl`* et *`John`*), *`Johnson`* (chef de projet de *`James`*, *`Ken`* et *`John`*).

- demander au programme de choisir aléatoirement un des 8 employés et faites-le travailler.

Amélioration : pour plus de clarté dans l'affichage, modifiez la méthode `Travaille()` en `Travaille(string tabulation)`. Cette tabulation sera utilisée devant chaque affichage Console : à chaque fois qu'un `ChefDeProjet` délègue à ses esclaves, il augmente la tabulation de 2 espaces, ce qui permet un affichage en «escalier», comme par exemple si on demande à *John* de travailler :

```
Je m'appelle John et je délègue le travail à mes esclaves.
  Je m'appelle Kévin et je commence à travailler.
  Je m'appelle Kévin et j'ai fini de travailler.
  Je m'appelle Jim et je délègue le travail à mes esclaves.
    Je m'appelle Kyle et je commence à travailler.
    Je m'appelle Kyle et j'ai fini de travailler.
    Je m'appelle Kévin et je commence à travailler.
    Je m'appelle Kévin et j'ai fini de travailler.
  Je m'appelle Jim et j'ai (enfin, mes esclaves...) terminé mon travail.
Je m'appelle John et j'ai (enfin, mes esclaves...) terminé mon travail.
```

EXERCICE 02_18

héritage
polymorphisme
design pattern
Strategy

Utilisez le *design pattern Strategy* pour implémenter une panier d'achats très simplifié comme proposée dans le diagramme de classes partiel ci-dessous.

Un `Article` possède un `Nom` et un `Prix`.

Un `MonPanier` peut être rempli à l'aide de la méthode `RemplirPanier` (puisqu'à ce moment du cours, nous n'avons pas encore vu les collections, vous pouvez utiliser un tableau d'`Article`s et en

conséquence, `RemplirPanier` écrasera le contenu de ce tableau à chaque appel, pour le remplacer par les `Articles` passés en paramètres).

Un `MonPanier` possède une stratégie de paiement (de type `IPaiementStrategy`) qui peut être modifiée à l'aide de la méthode `ChoisirStratégieDePaiement`.
La propriété calculée `PrixTotal` calcul le coût de `MonPanier` en utilisant la stratégie de paiement choisie.
Implémentez trois stratégies de paiement :
un `PaiementNormal` qui ne fait que calculer la somme des prix des `Articles` de `MonPanier`,
un `ParPalliers` qui applique une réduction sur le total de :
10% si `MonPanier` possède plus de 2 articles,
20% si `MonPanier` possède plus de 3 articles,
25% si `MonPanier` possède plus de 4 articles,
30% si `MonPanier` possède plus de 5 articles,
un `TVAOfferte` qui rembourse la TVA.
Testez vos classes en créant différents `MonPanier`s et en utilisant différentes stratégies de paiement.

EXERCICE 02_19

Format

Parse

Soit la classe `Personne` suivante :

```
public class Personne
{
	public Personne (string nom, bool porteDesLunettes,
	uint nbDentsPerdues, DateTime dateDeNaissance,
	TimeSpan tempsPerdu)
	{
		Nom = nom;
		PorteDesLunettes = porteDesLunettes;
		NbDentsPerdues = nbDentsPerdues;
		DateDeNaissance = dateDeNaissance;
		TempsPerdu = tempsPerdu;
	}

	public string Nom
	{  get; private set; }

	public bool PorteDesLunettes
	{  get; private set; }
```

```
        public uint NbDentsPerdues
        {  get; private set; }

        public DateTime DateDeNaissance
        {  get; private set; }

        public TimeSpan TempsPerdu
        {  get; private set; }
}
```

1) Écrivez une application Console permettant de construire une instance de `Personne` en posant les questions suivantes :

Question	Réponses autorisées	Exemple
`Comment vous appelez-vous ?`	n'importe quelle chaîne de caractères	`Bob l'Éponge`
`Portez-vous des lunettes ?`	true, false, True, False	`false`
`Combien de dents avez-vous perdues ?`	n'importe quel entier positif ou nul	`15`
`Quelle est votre date de naissance ?`	dates sous la forme : jj/mm/yyyy	`01/05/1999`
`Combien de temps avez-vous perdu ? (heures:minutes:secondes)`	temps sous la forme : hh:mm:ss	`16:13:54`

Vous devrez naturellement utiliser `TryParse` ou `Parse` sur les types `bool`, `uint`, `DateTime` et `TimeSpan` pour cela.

2) Ajoutez une méthode `ToString` à la classe `Personne` en utilisant un `StringBuilder` et les Format standards ou customs de `DateTime` et `TimeSpan` pour obtenir un résultat comme celui-ci (en reprenant l'exemple du tableau) :

```
Bob l'Éponge est né(e) le : samedi 1 mai 1999
Bob l'Éponge ne porte pas de lunettes
Bob l'Éponge a perdu 15 dents
Bob l'Éponge a perdu 16 heure(s) et 13 minute(s)
```