POO

des moules et des gâteaux

Consignes

Dans ce TP, vous ne devez rendre **uniquement** que les fichiers sources (`.cs`) de vos programmes.
La structure du rendu doit être comme ceci :

```
/rendu-tp-login_x.zip
   |-- login_x /
          |-- MultiSet.cs
          |-- BinaryTree.cs
          |-- Applications.cs
          |-- AUTHORS
          |-- README
```

`rendu-tp-login_x.zip`	nom de l'archive à rendre
`AUTHORS`	vous devez remplacer le contenu du fichier par une étoile `*`, un espace, puis votre login (ex : `login_x`) suivi d'un retour à la ligne
`README`	vous devez écrire dans ce fichier, tout commentaire sur le TP, votre travail ou bien plus généralement sur vos points forts/faibles. un fichier `README` vide sera considéré comme une archive sale (malus).

Il va de soi que toutes les occurrences de `login_x` sont à remplacer par **votre** login.

Tout le TP est à réaliser en mode console : `Fichier` ⇒ `Nouveau` ⇒ `Projet` ⇒ `Application Console`

TESTEZ VOTRE CODE !

In object-oriented programming, a class is an extensible template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions, methods). In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (subroutine that creates objects), and as the type of objects generated by the type, and these distinct concepts are easily conflated.
When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

– **wikipedia**

1 Introduction

Toutes les notions abordées dans cette section sont mises en pratique dans la partie annexes.

Pour le moment vous avez appris à déclarer des variables, utiliser des opérateurs pour créer des valeurs, appeler des méthodes, et écrire les principales instructions nécessaires lors de l'implémentation d'une méthode. Nous en savons largement assez pour passer à la vitesse supérieure.
Le Framework `.NET` contient des milliers de **classes**, dont vous avez déjà utilisé certaines d'entre elles, comme `Console` ou `Random`. Les **classes** fournissent un facilité de modélisation des éléments manipulés par vos applications. Ces éléments peuvent représenter quelque chose de spécifique, comme un personnage, une maison, ou quelque chose de plus abstrait, comme une force de gravité.

Lors de la conception d'une application, une grosse partie consiste à déterminer les éléments qui sont importants et à identifier leurs informations et comportements. Les informations stockées dans une **classe** sont appelées des **attributs**, et les comportements des **méthodes**.

1.1 Encapsulation

L'encapsulation est un principe important lors de la définition d'une classe. L'idée est que le programme qui utilise cette classe ne doit pas avoir à se soucier du fonctionnement interne de cette classe. Le programme va créer une **instance** de cette classe et appeler les **méthodes** de celle-ci. L'encapsulation est un mécanisme servant à dissimuler les informations et activités des différentes méthodes qui ne sont pas directement utile au programme qui l'utilise. On peut citer en exemple la méthode `Console.WriteLine()` dont on ne souhaite pas savoir comment la classe `Console` organise ses informations pour faire l'affichage demandé, ou du moins pas pendant le cycle prépa.

Par l'encapsulation on souhaite contrôler l'accessibilité des différentes **méthodes** et **attributs** de notre **classe**.

1.2 Visibilité

Lors de la déclaration d'une classe toutes ses méthodes et attributs ne sont pas accessibles depuis l'**extérieur** de notre classe. C'est-à-dire qu'une instance de notre classe ne peut ni voir les informations qu'elle contient, ni la faire fonctionner.
Nous avons cependant la possibilité de pouvoir modifier cette **visibilité** via les contrôleurs d'**accessibilités** : les mots-clefs `public` et `private`[1]. Pour changer la visibilité d'une **méthode** ou d'un **attribut** il suffit de le préfixer par le mot-clef adéquate.

Par défaut la visibilité est donc en privé (`private`).

Par convention la plupart des attributs sont en `private` et les méthodes en `public`. Si nous devons laisser l'utilisateur changer le contenu d'un attribut nous pouvons le placer en `public`, ou plus proprement, réaliser des méthodes permettant l'accès et la modification de celui-ci (**getter** et **setter**).
Visual Studio propose de les générer de façon automagique pour vous, il suffit de faire un clic droit sur l'attribut ⇒ `Refactor` ⇒ `Encapsulate Field`[2].

1. il en existe d'autres que je vous laisserai voir par vous même maintenant, ou que nous verrons plus tard ensemble.
2. `Ctrl+R, E`

1.3 Constructeur et instanciation

Lorsque nous souhaitons créer une nouvelle instance de notre classe, un **objet** donc, nous allons utiliser le mot-clef `new`. C'est lors de l'exécution de notre programme (**runtime**) que notre objet sera créé, une zone mémoire va lui être attribué et remplie en fonction de ses attributs et méthodes, puis initialisée en fonction de son **constructeur** si spécifié.

Un constructeur est une méthode spéciale qui s'exécute automatiquement lors de l'instanciation d'une classe. Il doit porter le même nom que la classe, peut prendre des paramètres, mais ne peut rien retourner. Chaque classe possède un constructeur par défaut (fourni par le compilateur) si aucun constructeur n'est spécifié, celui-ci ne fait absolument rien. Si vous écrivez votre propre constructeur, celui par défaut disparait. Selon les cas, il peut être intéressant, ou pas, d'avoir un constructeur sans paramètres. Il est également possible de **surcharger** un constructeur [3] en en spécifiant plusieurs, à partir du moment ou les paramètres sont différents (au niveau des types).

1.4 Méthodes et attributs `static`

Comme vous avez pu le voir avec la méthode `Main` de votre classe `Program`, celle-ci est en `static`. Le mot-clef `static` nous permet de déclarer des méthodes ou attributs accessible (en fonction de leurs visibilités) directement sans passer par une instanciation de la classe qui les contient.
Pour exemple la méthode `Sqrt` de la classe `Math` est en `static` ce qui nous permet de faire directement `Math.Sqrt()`. Nous nous dispensons de devoir instancier la classe `Math` pour utiliser sa méthode, ce qui est beaucoup plus pratique.

Lorsqu'on définit une méthode `static`, elle n'aura accès qu'aux attributs et méthodes `static` de la classe qui la contient. Les attributs déclarés en `static` sont donc commun à toutes les instances de cette classe.

On peut également définir une classe toute entière en `static`, celle-ci ne pourra donc pas être instanciée, et toutes ses méthodes et attributs devront être déclarés en `static` également.

1.5 Constantes

En préfixant un attribut du mot-clef `const`, vous pouvez spécifier que celui est `static` mais ne peut pas changer de valeur. Pour le moment on se limitera aux énumération, aux types numériques et aux chaînes.

1.6 Moi-même et rien

Dernières petites notions, le mot-clef `this`, permet à l'intérieur d'une méthode d'accéder à l'objet appelant cette méthode, cela peut être pratique en cas d'ambiguïté entre les attributs de la classe et les paramètres de la méthode, ou pour faire un constructeur par copie.
L'autre est le mot-clef `null` qui permet de dire qu'une valeur ne contient rien, celle-ci fonctionne à peu prêt avec tous les types [4].

3. toutes les méthodes d'une classe en fait
4. on peut faire un booléen à trois états (`True`, `False` et `null`)

2 Mises en pratique

Dans cette section nous allons vous demander d'implémenter vos types abstraits favoris [5]. Un rappel de la signature du type abstrait vous est donné à chaque fois, référez-vous au site [6] pour les détails de comportement des opérations. Un modèle de classe vous est également fourni, vous devez y compléter les parties manquantes.

On considèrera que le type `element`, utilisé dans les types abstraits, sera représenté par le type `string`. Il va de soi que vous n'avez pas le droit d'utiliser les classes pré-existantes `List`, `Set` ou autres du `C#` pour faire le boulot à votre place.

2.1 Des ensembles

Fichier à rendre : `MultiSet.cs`

Type abstrait

types
multiensemble

utilise
element, booleen, entier

operations

multiensemblevide	:	→ multiensemble
ajouter	:	element x multiensemble → multiensemble
supprimer	:	element x multiensemble → multiensemble
_ ∈ _	:	element x multiensemble → booleen
card	:	multiensemble → entier
nboccurrences	:	element x multiensemble → entier
choisir	:	multiensemble → element

Classe

```
class MultiSet
{
    /* fixme: attributs */

    /* visibility */ MultiSet()
        { /* fixme: multiensemblevide */ }

    /* visibility */ /* return type */ Insert(string element)
        { /* fixme: ajouter */ }
    /* visibility */ /* return type */ Delete(string element)
        { /* fixme: supprimer */ }
    /* visibility */ /* return type */ Belong(string element)
        { /* fixme: appartient */ }
    /* visibility */ /* return type */ Size()
        { /* fixme: card */ }
    /* visibility */ /* return type */ NbOccurences()
        { /* fixme: nboccurrences */ }
    /* visibility */ /* return type */ Pop()
        { /* fixme: choisir */ }
}
```

5. on le sait !
6. `http://algo.infoprepa.epita.fr/index.php`

2.2 Des arbres

Fichier à rendre : `BinaryTree.cs`

Type abstrait

```
types
   arbrebinaire

utilise
   noeud, element

operations
   arbrevide   :  → arbrebinaire
     <_,_,_>   :  noeud x arbrebinaire x arbrebinaire → arbrebinaire
      racine   :  arbrebinaire → noeud
           g   :  arbrebinaire → arbrebinaire
           d   :  arbrebinaire → arbrebinaire
     contenu   :  noeud → element
```

Classe

```
class BinaryTree
{
    /* fixme: type */ _left;
    /* fixme: type */ _right;
    /* fixme: attributs */

    /* visibility */ BinaryTree() { /* ? */ }
    /* visibility */ BinaryTree(string             element,
                                /* fixme: type */ left,
                                /* fixme: type */ right) { /* ? */ }

    /* visibility */ /* fixme: type */ Left {
        get { /* fixme */ }
        set { /* fixme */ }
    }

    /* visibility */ /* fixme: type */ Right {
        get { /* fixme */ }
        set { /* fixme */ }
    }

    /* visibility */ /* return type */ Root() { /* fixme */ }
}
```

3 Applications

Fichier à rendre : `Applications.cs`

Dans cette section nous allons utiliser les classes écrites précédemment. Pour cela vous implémenterez les méthodes suivantes (en `static`) dans la classe `Applications`.

```
class Applications
{
    /* MultiSet */
    public static void MultiSetDisplay(MultiSet set) { }

    /* BinaryTree */
    public static void DepthFirstDPreorder(BinaryTree tree) { }
    public static void DepthFirstDInorder(BinaryTree tree) { }
    public static void DepthFirstDPostorder(BinaryTree tree) { }
    public static void BreadthFirstDisplay(BinaryTree tree) { }

    /* Both */
    public static MultiSet Convert(BinaryTree tree) { }

    /* Bonus */
    public static BinaryTree AddLeaf(BinaryTree bst, string elt) { }
    public static BinaryTree Convert(MultiSet set) { }
}
```

- `MultiSetDisplay(MultiSet)` : affichage complet des éléments du multiensemble
- `DepthFirstDisplayPreorder(BinaryTree)` : affichage parcours en profondeur ordre prefixe
- `DepthFirstDisplayInorder(BinaryTree)` : affichage parcours en profondeur ordre infixe
- `DepthFirstDisplayPostorder(BinaryTree)` : affichage parcours en profondeur ordre suffixe
- `BreadthFirstDisplay(BinaryTree)` : affichage parcours en largeur
- `Convert(BinaryTree)` : convertir un arbre binaire en multiensemble
- `AddLeaf(BinaryTree, string)` : ajout en feuille dans un arbre binaire de recherche
- `Convert(MultiSet)` : convertir un multiensemble en arbre binaire de recherche

Annexes

Déclaration

```
class Rectangle {
    private double _width;                // attribut
    private double _length;               // attribut
    private static int _count = 0;        // static attribut
    private const int _d = 2;             // constant value

    public double Width                   // encapsulate
    {
        get { return _width; }            // _width getter
        set { _width = value; }           // _width setter
    }
    public double Length                  // encapsulate
    {
        get { return _length; }           // _length getter
        set { _length = value; }          // _length setter
    }
    public static int Count               // encapsulate
    {
        get { return _count; }            // _count getter
        set { _count = value; }           // _count setter
    }

    public Rectangle() { }                // default constructor
    public Rectangle(double w, double l) // overload constructor
    {
        this._width = w;
        this._length = l;
        ++_count;
    }

    public double Area()                  // area method
    {
        return _d * _width + _d * _length ;
    }
}
```

Utilisation

```
class Program {
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle();    // new instance by default
        Rectangle d = new Rectangle(2,3); // new instance

        r.Width = 4;                      // set _width to 4
        r.Length = 8;                     // set _length to 8

        Console.WriteLine(r.Area());          // display 24
        Console.WriteLine(d.Area());          // display 10
        Console.WriteLine(Rectangle.Count);  // display 1
        Console.ReadKey();
    }
}
```