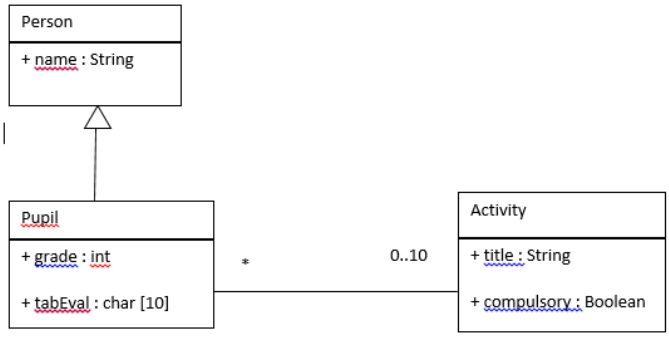
C# : Synthèse

1. Contexte

Soit le diagramme de classes UML ci-dessous. Des enfants de primaire peuvent choisir des activités au cours de la semaine et obtenir une évaluation. 10 activités maximum peuvent être choisies par chaque enfant. Pour l'enfant, on garde trace aussi de l'année dans laquelle il est. Certaines activités sont obligatoires.



2. Construction d'une classe

Les méthodes de sélection (get) et de modification (set) ne sont plus d'application !
Veuillez sélectionner le mot « name » dans la déclarative de cet attribut, choisissez les menus « Edit ; Refactor[1] ; Encapsulate Field » et cliquez sur le bouton OK. Suivez en lisant attentivement, en ne modifiant rien et cliquez sur Apply.

```
class Person
{
    //private String name;
    private int age;

    //obtenu via Edit ; Refactor  ; Encapsulate Field
    //seulement si on veut une vérification du nom
    /*public string Name{
        get { return name;}

        set{ name = value;}
    }*/

    //raccourci si pas envie de mettre de vérification du nom
    //avec cette version on doit virer la declaration de la variable que j'ai du coup mis en comm au dssu
    public String Name { get; set; }
    public int Age
    {
        get { return age; }
        set { age = (value > 0) ? value : 1; }
    }

    public Person(String name, int age)
    {
        Name = name;
        Age = age;
    }
```

Name est ce qu'on appelle une "**Property**" ou propriété. **Notion très importante !**

Ce qui influence l'écriture du code par la suite.
Value est semblable à un paramètre d'entrée sur une méthode. Le mot **value** référence la valeur que le code client essaie d'assigner à la propriété.

[1] Qu'est-ce que le refactoring ?

Plutôt que d'écrire otherName = this.getName() ; ou otherName = getName(); ou otherName = objectX.getName();, **on utilisera la propriété et on écrira à droite de l'affectation** : name = **Name** ;	Plutôt que d'écrire objectX.setName("riri") ; **on utilisera aussi la propriété et on écrira :** objectX.**Name** = "riri" ;

« Le refactoring est l'opération consistant à retravailler le code source d'un programme informatique – sans toutefois y ajouter des fonctionnalités ni en corriger les bogues – de façon à en améliorer la lisibilité et par voie de conséquence la maintenance, ou à le rendre plus générique (afin par exemple de faciliter le passage de simple en multiple précision) ; on parle aussi de « remaniement ». Cette technique utilise quelques méthodes propres à l'optimisation de code, avec des objectifs différents. »

Si on ne souhaite pas valider les entrées de « value », on n'écrit pas tout cela. public String Name {get ; set ;} et de retirer le reste **y compris** la déclarative de l'attribut *private String name ;*

```
//raccourci si pas envie de mettre de vérification du nom
//avec cette version on doit virer la declaration de la variable que j'ai du coup mis en comm au dssu
public String Name { get; set; }
public int Age
{
    get { return age; }
    set { age = (value > 0) ? value : 1; }
}

public Person(String name, int age)
{
    Name = name;
    Age = age;
}
```

Ajoutons un constructeur.

```
public Person(String name, int age)
{
    Name = name;
    Age = age;
}
```

Ajoutons la méthode ToString pour afficher le message : *… âgé(e) de … ans*
Elle fonctionne comme en Java excepté le fait que son en-tête sera :
public **override** string ToString() { … } où le mot override signifie que la méthode ToString déclarée « **virtual** » dans la super-classe mère

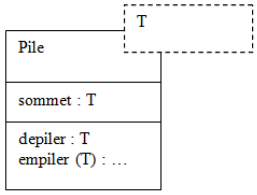
NB : l'héritage lui se fait via le symbole « : ».

NB2 : Attention qu'a l'écriture ici le super.Parent pour appeler le constructeur parent, le mot base est utilisé à la place.

Construction d'une sous-classe - Classe générique - Tableau statique

Avant de préciser les attributs de la classe Pupil dans votre code, signalons un élément important dans un développement informatique : les collections génériques. Ce sont des collections qui sont typées au moment de la compilation. Parmi les collections génériques .net,
Queue<T>, Stack<T>, LinkedList<T>, ConcurrentQueue<T>, ConcurrentStack<T>
List<T>, Dictionary<TKey,TValue>, SortedDictionary<TKey,TValue>
etc. Les éléments T, TKey, TValue seront typés à la compilation.

A titre d'exemple, la classe Pile où T est le paramètre typé; le schéma UML[2] est le suivant :



Pour la collection List, la propriété est marquée **internal** via l'encapsulation. Modifiez-la en public. 5 types d'accès existent : public (accès non limité), private (accès dans la classe), protected (accès dans la classe ou les classes-fille), internal (accès limité à l'assembly[3]) et protected internal (accès limité à l'assembly ou aux classes-fille).

[2] Cf. activité d'enseignement PPOO IG1

[3] "*Un assembly est un bloc de construction fondamental de toute application .NET Framework. Par exemple, lorsque vous générez une simple application C#, Visual Studio crée un assembly sous la forme d'un unique fichier exécutable portable (PE, Portable Executable), spécifiquement un EXE ou DLL. Les assemblys contiennent des métadonnées qui décrivent leur propre numéro de version interne et les détails de tous les types de données et d'objet qu'ils contiennent*." (http://msdn.microsoft.com/fr-fr/library/ms173099(v=vs.80).aspx)

3. Paramètres nommés

On peut donner lors des méthodes des valeurs par défaut aux paramètres entrés.

```
public void AddEvaluation(String title = null, Parameter.Eval eval = Parameter.Eval.S)
{
    pupilEvaluation[Indice] = (char)eval;
    this.indice++;
}
```

Cette activité sera évaluée par défaut égale à S ! C'est ainsi que chaque fois que la méthode sera appelée si un des arguments n'est pas donné à l'appel on utilisera une valeur par défaut.

4. Variable anonyme

La variable indice ci-dessus aurait pu être déclarée : **var** i = 0;
Dans ce cas, elle est anonyme et lors de la compilation, prendra le type donné à l'initialisation. La variable i sera ainsi de type int.
Nous pouvons agir ainsi pour d'autres éléments. Ce qui donne dans les cas ci-dessous,

Créez une liste avec quelques élèves de diverses années primaires et de divers âges. Garnissez une **variable anonyme** pupilGrade1Plus6 avec les enfants qui font partie de 1e année et qui ont plus de 6 ans. Imprimez les enfants qui en font partie.

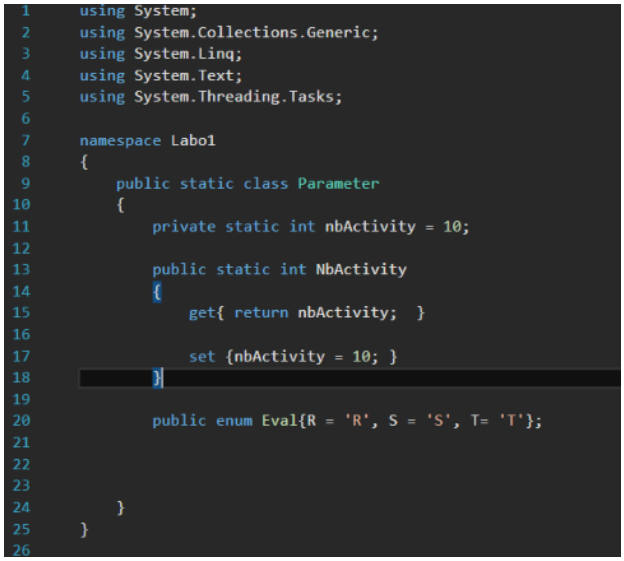
```
/* ==> première version avec LINQ*/
var pupilGradePlus6 = from pupil in listPupil
                      where pupil.Age > 6 && pupil.Grade == 1
                      select pupil;
//notons ici que le select est en dernier... pq?
```

Classe statique

- non instanciable ; impossible de construire un objet new ...
- contient uniquement des membres statiques
- est de type **sealed** (ne peut donc être héritée)
- ne peut hériter d'aucune classe sauf Object
- ne peut contenir de constructeur d'instance mais peut contenir un constructeur statique

Conteneur pour les ensembles de méthodes qui opèrent sur des paramètres d'entrée et n'ont pas à obtenir ou à définir de variables d'instance.

Créez une classe statique appelée Parameter.cs dans votre projet.
Elle comprendra une constante symbolique correspondant au nombre 10 qui est le maximum d'activités qui peuvent être choisies par un élève.
Adaptez la classe Pupil.cs.
Créez aussi une énumération pour les évaluations 'R', 'S' et 'T'. Adaptez votre code également.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Labo1
{
    public static class Parameter
    {
        private static int nbActivity = 10;

        public static int NbActivity
        {
            get{ return nbActivity;  }

            set {nbActivity = 10; }
        }

        public enum Eval{R = 'R', S = 'S', T= 'T'};



    }
}
```

5. Structures

Dans certains cas où les services offerts par l'écriture de classes sont inutiles (aucune méthode n'est nécessaire, aucun héritage, ne pas vouloir le runtime .Net), le programmeur peut utiliser une **structure ("classe en réduction").**

Exemple :

```
struct Dimensions
{
        public double Length, Width;
}
Dimensions point;
point.Length = 10;
```

Les structures sont stockées soit dans la pile soit "inline"; elles ont les mêmes restrictions de durée de vie que les types simples de données.

Les structures sont plus limitées que les classes.

- Dans une déclaration de structure, les champs ne peuvent pas être initialisés sauf s'ils sont déclarés const ou static.
- Une structure ne peut pas déclarer de constructeur par défaut (un constructeur sans paramètres).
- Elle est copiée lors de l'assignation. Lorsqu'une structure est assignée à une nouvelle variable, toutes les données sont copiées et les modifications apportées à la nouvelle copie ne changent pas les données de la copie d'origine.
- Une structure est de type valeur (une classe est de type référence).
- Une structure peut être utilisée comme un type Nullable et peut se voir assigner une valeur Null.
- Contrairement à une classe, un objet struct peut être instancié sans recours à l'opérateur new.
- Une structure ne peut pas hériter d'une autre structure ou d'une classe ; elle ne peut pas être héritée.
  Elle hérite directement de *System.ValueType*, qui hérite de *System.Object*.
- Une structure peut implémenter des interfaces.

6. Expression lambda

Une expression lambda est une **fonction anonyme (fonction qui ne porte pas de nom et qui est encodée directement à l'endroit souhaité)** qui peut contenir des expressions et des instructions, et qui peut être utilisée pour créer des délégués[4] ou des types d'arborescence d'expression.

Syntaxe

(variable d'entrée, variable d'entrée, ...) => {instruction1; instruction2; ...}

où le symbôle => se lit "conduit à", les parenthèses sont obligatoires quand il n'y a pas d'entrée.

```
// version expression lambda. on saura ici que pupil est un objet de la liste car la liste est de type pupil
var pupilGradePlus6 = listPupil.Where(pupil => pupil.Age > 6 && pupil.Grade == 1);
```

[4] Notion vue au cours d'un prochain laboratoire ou cours de théorie

7. Covariance

La covariance permet la **conversion implicite** des références pour des types de tableau, les types délégués (notion autre laboratoire) et les arguments de type générique.

Appliquons ces notions à notre projet. Dans main, créez une liste d'étudiants appelée listPupils.

créez une liste de personnes listPersons (même démarche),
fusionnez les deux listes : listPersons et listPupils en une liste anonyme listFusion
var listFusion = listPersons.Union(listPupils) ; // grâce à la covariance et
faites afficher les noms de chaque membre de la liste complète.

```
    //********** déclarations***********
    List<Pupil> listPupilDuplicated = new List<Pupil>() {

        new Pupil("Manon", 6, 1),
        new Pupil("Manon", 6, 1),
        new Pupil("Adrien", 7, 1),
        new Pupil("Adrien", 7, 2),
        new Pupil("Louis", 6, 1),
        new Pupil("Adrien", 7, 2),
        new Pupil("Marvin", 8, 1),
};

    Pupil manon = new Pupil("manon", 6, 1);
    List<Pupil> listPupil = new List<Pupil>() {

        new Pupil("Manon", 6, 1),
        new Pupil("Nicolas", 9, 1),
        new Pupil("Adrien", 7, 1),
        new Pupil("Jennifer", 6, 2),
        new Pupil("Louis", 6, 1),
        new Pupil("Vale", 7, 2),
        new Pupil("Marvin", 8, 1),
};


    List<Person> listPerson = new List<Person>() {

        new Person("Manon", 6),
        new Person("Nicolas", 9),
        new Person("Adrien", 7),
        new Person("Jennifer", 6),
        new Person("Louis", 6),
        new Person("Vale", 7),
        new Person("Marvin", 8),
};
        // union via le principe de covariance
    var listFusion = listPerson.Union(listPupil);
```

Une interface est une classe abstraite sans attributs qui ne comprend que des méthodes abstraites *(et par conséquent, aucun constructeur) et qui peut accepter des définitions de constantes. Dans un développement de logiciels, elle sert à signaler aux autres programmeurs de l'équipe les méthodes utiles à tous.*

Dans main, créez une liste d'élèves avec des doublons point de vue nom et âge : List<Pupil> listPupilsDuplicated….
pour créer une liste sans redondance,
implémentez dans une nouvelle classe ***PersonComparer*** *qui hérite de l'interface IEqualityComparer : public class PersonComparer : IEqualityComparer<Person> { … }*
écrivez les deux méthodes de cette interface Equals et GetHashCode[5]
la méthode Equals renvoie une variable de type Boolean et reçoit deux arguments qui sont des personnes dans notre contexte.
Elle renvoie true si les deux arguments pointent vers la même instance ou s'ils ont même nom et âge.
la méthode GetHashCode renvoie un entier, résultat du hashcode du nom de la personne ^ âge de la personne.
créez la liste sans redondance : IEnumerable<Pupil> listPupilsNoDuplicated = listPupilsDuplicated.Distinct<Pupil>(new PersonComparer());
faites imprimer le nombre de personnes de cette liste.

[5] A vous de chercher comment implémenter ces deux méthodes

8. Passage par référence

Par défaut, certains arguments passés à une méthode passent en copie. Si on souhaite les passer par adresse, le mot « **ref** » existe.

.... methode (int x)
{
x = 10;
}
 	
Appel :
int x = 15;
....methode(x);
// x vaut toujours 15
	.... methode (**ref** int x)
	{
	x = 10;
	}
	
	Appel :
	int x = 15;
	....methode(x);
	// x vaudra 10

9. Divers

Dans la méthode ToString de la classe Pupil, créez **deux méthodes privées** la première correspondant à tout ce qui est en-tête sans la liste des activités : via la sélection du texte, Edit, Refactor, Extract Method. Nom de la méthode : Header
la seconde PrintActivities englobant le code qui affiche l'ensemble des activités
pour que la fonction ressemble à :

```
public override string ToString()
{
    string chaine = HeaderPupil();
    return printActivities(chaine);
}

private string printActivities(string chaine)
{
    if (ListActivities.Count() == 0) { chaine += " n'a pas encore choisi"; return chaine; }
    else
    {
        chaine += "a choisi les activités suivantes : \n";
        foreach (Activity activity in ListActivities)
        {
            chaine += activity.ToString() + "\n";
        }
        return chaine;
    }
}

private string HeaderPupil() { return base.ToString();}

public void AddEvaluation(String title = null, Parameter.Eval eval = Parameter.Eval.S)
{
    pupilEvaluation[Indice] = (char)eval;
    this.indice++;
}
```

10.Dictionnaire

Retirons la liste des activités et le tableau des évaluations pour en faire un **Dictionary**[6]. Ensemble d'objets qui comprendront le titre de l'activité (identifiant de l'activité dans notre exercice, servant de clé au dictionnaire) et l'évaluation obtenue pour l'activité.

```
private Dictionary<String, char> pupilActivities = new Dictionary<String, char>();

public Dictionary<String, char> PupilActivities {get;set;}

public void AddActivity(String activityTitle)
{
        PupilActivities.Add(activityTitle, 0);
          }
```

[6] Quelle est l'efficacité de ce type d'organisation ?

```
public void AddEvaluation(String title = null, char evaluation='S')
{
        if (title != null)   PupilActivities [title] = evaluation;
          }

private string PrintActivitiesPupil(string ch)
{
          …
             ch += "\n" + PupilActivities.ElementAt(i).Key.ToString() + " : " +
                                                          PupilActivities.ElementAt(i).Value ;
          return ch;
}
```

Pour tester (et ne pas perdre de temps), placez en commentaires les méthodes que vous ne souhaitez pas rectifier ainsi que beaucoup d'instructions dans main.