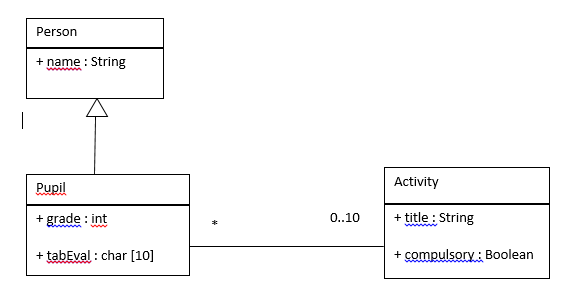
|  |  |
| --- | --- |
| M:\ADMIN\henallux_montgolfiere.png  Implantation IESN | C:\Users\DubyLocal\Documents\ABRacine\Coordination\Logo Henallux\IG-v1.1-R1.jpg**Environnement de développement de logiciels**  IG3 — Labo 1 — 2016-2017 |

|  |
| --- |
| **Objectifs**   * Notions de base en C# et en LINQ   + Solution avec un projet comportant plusieurs classes   + Concept : espace de noms   + Héritage   + Notion de propriété   + Collection générique   + Tableau   + Paramètres nommés   + Passage par référence   + Variable anonyme   + Types de classes   + Expression lambda   + Covariance * Refactoring * Nous insistons sur la standardisation du code : une méthode commence par une majuscule, on utilise les propriétés, une variable porte un nom soit anglais soit français, etc… (cf. Clean Code) |

**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.



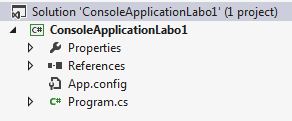
**Création du projet de type Console**

Lancez VisualStudio. Soit vous avancez via la « start page » soit vous fermez cet onglet et vous procédez via le menu File.

Vous choisissez New Project. Vous créez une solution (Installed; Templates; VisualC# ; Windows Desktop; ConsoleApplication). Son nom sera ConsoleApplicationLabo1. Vérifiez que « Create directory for Solution » soit bien coché et que le chemin d'accès est celui souhaité.

Vous arrivez à un écran composé de plusieurs parties.

Dans la fenêtre Solution Explorer, vous y verrez le nom de la solution ainsi que le premier projet inclus dans celle-ci. Une solution est un ensemble de projets qui peuvent être écrits dans des langages distincts.



Invitation à parcourir l'arborescence et à découvrir ce qui s'y cache.

Votre projet comprend une classe par défaut qui se nomme Program.cs. Ce sera la classe qui servira aux créations d’objets et aux différents appels de méthodes.

**Construction d'une classe**

Ajoutez une nouvelle classe Person.cs. Soit se placer dans Solution Explorer, sur le projet. Click droit. Ajoutez un item à ce projet de type Class appelée Person.cs.

Soit Project Add New Item ….

Remarquons déjà :

* **using**
* **namespace**

Soit le premier attribut :   
 private String name;  
Déjà deux solutions string ou String… string est un alias de la classe String ; la comparaison des chaines déclarées de type string se fait par égalité et non via equals. Remarquez aussi que l'attribut commence par une minuscule.

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]](#footnote-1) ; Encapsulate Field » et cliquez sur le bouton OK. Suivez en lisant attentivement, en ne modifiant rien et cliquez sur Apply.  
Vous obtenez :

public String **Name** {   
 get {return name;}   
 set {name = value;}  
 }

Name est ce qu’on appelle une “**Property**” ou propriété. **Notion très importante !**

**Ce qui influence l’écriture du code par la suite.**

Le mot clé **value** est utilisé dans l'accesseur set. Il 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.

Plutôt que d’écrire   
 otherName = this.getName() ;   
ou   
 otherName = getName();   
ou   
 otherName = objectX.getName();

comme en Java,

**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" ;

Agissez de même pour l’attribut âge. Ce qui donne :

public int **Age** {   
 get {return age;}   
 set {age = value;}  
 }

Si on souhaite valider l'entrée qui se fait via « value » (exemple : un âge doit être positif), on pourrait écrire :

public int **Age** {   
 get {return age;}   
 set {age = (value > 0) ? value : 1 ;}  
 }

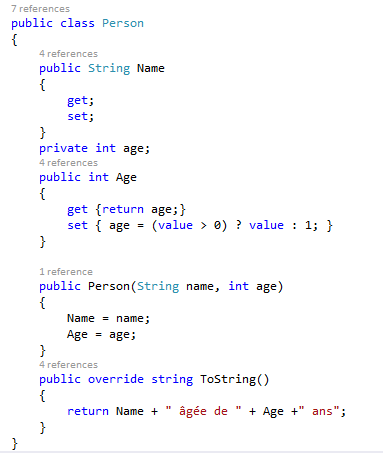
**Si on ne souhaite pas valider les entrées de « value », on n’écrit pas tout cela.**   
Essayons pour le nom.   
Il suffit d’encoder   
 public String Name {get ; set ;}   
et de retirer le reste **y compris** la déclarative de l’attribut *private String name ;*.   
**Habituez-vous à ce type d’écriture !**

Ajoutons un constructeur.   
On obtient :

public Person (String name, int age) {  
 Name = name ; // On emploie **toujours** la propriété  
 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 Object est supplantée par celle de la classe Person.

Votre classe est donc :



**Construction de la classe Activity**

La classe Activity.cs comporte 2 attributs (page 1); placez au moins un constructeur et la méthode d’affichage suivante … (…)  
  
 libellé de l’activité message *obligatoire* si l’activité est obligatoire

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

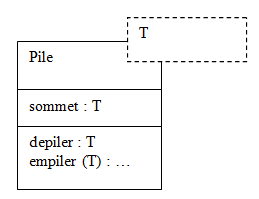
Nous abordons la classe Pupil.  
Signature de la classe fille : Class Pupil **: Person**  
Signification des « : » : Pupil hérite de Person .

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.   
Ces collections ont des attributs et méthodes déjà existantes. Il est également possible d'écrire ses propres classes génériques.

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



Soient les attributs de la classe Pupil :

* grade : entier
* liste d’activités : private **List<Activity>** lstActivities ;
* tableau d’évaluations (nous supposons que les évaluations seront en correspondance avec l’activité choisie et que l'élève ne peut prendre que 10 activités maximum) :   
   private **char []** tabEval ;  
  L’évaluation vaudra R(ecommencez), S(atisfaisant), T(rès bon).

N'oubliez pas les propriétés correspondantes.  
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]](#footnote-3)) et protected internal (accès limité à l'assembly ou aux classes-fille).

Constructeurs

* un premier constructeur qui servira au deuxième : l'élève n'a pas choisi d’activité.  
   public Pupil (String name, int age, int grade) : **base**(name,age) {   
   Grade = grade ;  
   LstActivities = new List<Activity>() ;

TabEval = new char [10] ;   
 }  
Le mot **base** signifie qu'on fait appel à un constructeur de la classe mère (ici Person).

* un second constructeur pour un élève de 1e primaire ( pas d’activité choisie)   
   public Pupil (String name, int age) : **this**(name,age, 1) {}  
  Le mot **this** référence le constructeur décrit auparavant (overriding).

Méthodes

* AddActivity : ajout d’une activité (pas de cas d’erreur à traiter)
* Impression des coordonnées de l’élève et des activités qu’il entreprend sous la forme

*… âgé(e) de … ans a choisi les activités suivantes :[[4]](#footnote-4)  
… (…)   
…*   
Ce qui donne le code suivant :  
public override string ToString()  
{  
 string ch = base.ToString() ;   
 int cptActivities = LstActivities.Count() ;   
 if (cptActivities == 0 )   
 {

ch += "n’a pasencore choisi d’activité");

}  
 else  
{  
ch +**= …. ;**

foreach(Activity activity in LstActivities)  
 ….  
}

**Test via la fonction main**

Dans la classe de test - à savoir *Program.cs*, créez un objet de type Pupil de 1e primaire.

Créez quelques objets de type Activity (obligatoires ou non).

Ajoutez ces activités à l’élève. Pour l'instant, aucune évaluation.

Imprimez tout ce qui concerne l’élève.  
 System.Console.Write(…);  
L’instruction   
 System.Console.Read();   
vous permettra de temporiser.  
  
L'environnement de développement permet de tester autrement et ce, de manière plus professionnelle. Ce sera vu ultérieurement.

**Paramètres nommés**

Construisez dans la classe Pupil une méthode AddEvaluation qui, pour une activité dont on donne le titre et l’évaluation, met à jour le tableau des évaluations. On donne des valeurs par défaut aux paramètres entrés.  
Agissez comme ci-après.

public void AddEvaluation ( **String title = null, char evaluation = ‘S’**) {  
 …

} // aucun cas d’erreur n’est envisagé  
  
Dans main, ajoutez à l'élève une évaluation à une activité choisie sans placer le deuxième argument. Cette activité sera évaluée par défaut égale à S !   
Illustrons par un exemple.  
Imaginons l'objet pupilGrade1, instance de Pupil auquel on a ajouté l’activité de coloriage :  
 Activity coloriage = new Activity("Coloriage",true);  
 pupilGrade1.AddActivity(coloriage);  
 pupilGrade1.AddEvaluation("Coloriage");  
Il aura ‘S’ comme évaluation.  
  
Ajoutez une évaluation à une activité en inversant les arguments de la manière suivante :  
 … . AddEvaluation (**evaluation :** ‘T’, **title :** "…");  
Pas de problème ! Ceci uniquement dans le cas où les paramètres sont nommés.  
  
Signalons qu'on peut toujours écrire :  
 pupilGrade1.AddEvaluation("Coloriage",’R’);

**Variable anonyme**

La variable i 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,  
Exemple  
 var p = new Person(…) ; // Référence vers un objet de type Person

Une variable anonyme peut recevoir le résultat de requêtes de données dans des structures de données.  
Exemple

List<Player>lstPlayers = new List<Player>()  
 {  
 new Player ("Player1", 12, 5),  
 new Player ("Player2", 0, 2),  
 };  
 où  
  
 Player est une classe représentant un joueur (attributs : name, victories, defeats).

On souhaite sélectionner, via une requête, les joueurs qui ont remporté des victoires et n'ont subi aucune défaite. Pour ce faire, le **langage LINQ** (Language INtegrated Query)[[5]](#footnote-5)  
est intégré dans l'environnement .net.  
L'instruction s'écrit :   
 var bestPlayers = **from** player **in** lstPlayers  
 **where** player.Victories > 0 **&&** player.Defeats ==0   
 **select** player;  
Interprétation : pour tout joueur (player) de la liste (lstPlayers) pour lequel le nombre de victoires est positif (where ...) et le nombre de défaites est nul, on cumule dans bestPlayers les objets player qui respectent la condition.  
Ensuite, on peut exploiter bestPlayers comme ci-dessous :

if(bestPlayers != null)   
 **foreach** ( var player in bestPlayers)  
 {   
 ...;  
 }   
 L'ordinateur sait exploiter chaque objet player de bestPlayers.

Appliquons cela dans la fonction main.  
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.

**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.

Exemple 1

La classe [*System.Math*](http://msdn.microsoft.com/fr-fr/library/system.math(v=vs.100).aspx) est statique et contient plusieurs méthodes qui exécutent des opérations mathématiques, sans qu'il soit nécessaire de stocker ou extraire des données uniques à une instance particulière de la classe [*Math*](http://msdn.microsoft.com/fr-fr/library/system.math(v=vs.100).aspx).

Exemple 2

public **static** class SortAlgorithm

{

public static void BubbleSort (...) {...}

public static void HeapSort (...) {...}

}

Appel : SortAlgorithm.BubbleSort(...);

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.

**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.

**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[[6]](#footnote-6) 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.

Exemple1

( x ) => x\*x x est la variable d'entrée et conduit au calcul de x\*x

Cette expression peut aussi s'écrire : x => x\*x

Exemple2

( ) => Console.WriteLine("Bonjour à tous!"); On est amené à imprimer.

Exemple 3

int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

int oddNumbers = numbers.Count ( n => n % 2 == 1);  
 // oddNumbers = nombre d'éléments impairs dans numbers

var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);  
 // on liste tous les nombres inférieurs à 6

Exemple 4

lstPersons.Where(person => person.Age >=18);  
 // C# sera capable de déterminer que person est un objet de type Person   
 // si lstPersons est une liste d'objets de type Person

Exemple 5

(int x, string s) => s.Length > x

// on considère s à condition que sa longueur soit supérieure à x

Dans l'exercice, modifiez le garnissage de la variable anonyme pupilGrade1Plus6.

**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.

Exemple 1

String chaine = "Informatique";  
Object obj = chaine;

Un objet instancié à partir d'une classe située plus bas dans la hiérarchie de classes (ici String) est affecté à un objet instancié à partir d'une classe située plus haut dans la hiérarchie (ici : Object)

Exemple 2

void Abreuver ( IEnumerable <Animal> animaux )

{

foreach ( Animal a in animaux)   
 a.AbreuverUn();  
}

...  
List <Ours> lstOurs = ...; // Ours est une sous-classe de Animal

....Abreuver(lstOurs);

Appliquons ces notions à notre projet.

Dans main,

* créez une liste d’étudiants appelée listPupils de la manière suivante:  
   List <Pupil> listPupils = new List <Pupil>()  
   {  
   new Pupil("AAAA", 10,4),   
   etc…  
   } ;
* 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.

Rappel pour ci-dessous: 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[[7]](#footnote-7)
    - 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.

**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 |

**Debugging :**Exemple :pour indiquer dans le code qu’on souhaite vérifier une valeur dans une fenêtre de l’environnement de développement Visual Studio, Debug.WriteLine(pupilGrade1Plus6);

**Clean Code? "No Bad Smells"?**Modifions le **nom de l'attribut** tabEval en pupilEvaluations. Ajustez votre code.

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 ch = HeaderPupil(); // appel de la 1e méthode  
 ch = PrintActivitiesPupil(ch); // appel de la seconde méthode  
 return ch;

}

Retirons la liste des activités et le tableau des évaluations pour en faire un **Dictionary[[8]](#footnote-8)**. 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);

}

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.

1. Qu’est-ce que le refactoring ? [↑](#footnote-ref-1)
2. Cf. activité d’enseignement PPOO IG1 [↑](#footnote-ref-2)
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) [↑](#footnote-ref-3)
4. Si l’élève n’a pas choisi d’activité, adaptez votre message [↑](#footnote-ref-4)
5. Ce langage permet d'interroger des bases de données, des fichiers XML ou autres flux de données. Il est intégré à la programmation .Net. [↑](#footnote-ref-5)
6. Notion vue au cours d'un prochain laboratoire ou cours de théorie [↑](#footnote-ref-6)
7. A vous de chercher comment implémenter ces deux méthodes [↑](#footnote-ref-7)
8. Quelle est l’efficacité de ce type d’organisation ? [↑](#footnote-ref-8)