Implantation IESN

Environnement de développement de logiciels

IG3 — Labo 2 — 2016-2017

Objectifs

- Appliquer ce qui a été vu dans le premier labo
- Notions avancées en C#
  - Dynamic
  - Délégués

Première partie : application des notions préalablement assimiliées

Soit une classe **Person**. Cette classe sera abstraite (on ne sait l'instancier directement - créer d'objets). Elle est déclarée :

public abstract Class Person { ... }.

Elle comprend les attributs name et lastname.

Dans cette classe, vous y placez

- les propriétés correspondantes,
- un constructeur qui reçoit les valeurs pour les deux attributs (même si nous ne pouvons instancier la classe, ce constructeur servira dans les sous-classes),
- une méthode ToString qui renvoie la chaîne composée du nom et du prénom,
- une méthode abstraite *HasHisBirthday* comme écrit ci-dessous :
  public abstract bool *HasHisBirthday* ();

Rappel : une classe abstraite est une classe non instanciable qui peut avoir une à plusieurs méthodes abstraites.

Soit une sous-classe de Person : **PrivateContact**. On y ajoute le numéro de téléphone privé, l'adresse mail privée ainsi que la date de naissance (vide si elle n'est pas connue). Pour la variable d'instance date de naissance, référez-vous à des classes existantes.

Dans cette classe, vous y placez

- les propriétés correspondantes,
- un constructeur qui reçoit les valeurs pour tous les attributs,
- un constructeur qui reçoit les valeurs pour tous les attributs sauf la date de naissance,
- une méthode ToString qui renvoie une chaîne sous la forme
  nom prénom (numéro de téléphone privé)
  adresse mail privée
- l'implémentation de la méthode *HasHisBirthday*
  public override bool *HasHisBirthday* ()
  {
  return (DateTime.Today.Month == DateNaiss.Month && ....);
  }

Créez deux contacts privés dans la classe **Program**, le premier sans date d'anniversaire connue et l'autre qui aura son anniversaire aujourd'hui.
Imprimez pour celui qui a son anniversaire aujourd'hui, la chaîne créée par ToString et le fait de lui dire "Bon anniversaire"!

Soit une sous-classe de Person : **ProfessionalContact**. On y ajoute la profession exercée par le contact professionnel, son numéro de téléphone professionnel et son adresse mail professionnelle.

Dans cette classe, vous y placez

- les propriétés correspondantes,
- un constructeur qui reçoit une valeur pour tous les attributs,
- une méthode ToString qui prévoit l'affichage de toutes les valeurs des attributs sous la forme:
  nom prénom (numéro de téléphone professionnel)
  profession
  adresse mail professionnelle
- l'implémentation de la méthode *HasHisBirthday* : elle renvoie toujours false.

Créez trois contacts professionnels dans la classe Program, deux consultants et un indépendant.

Le contact professionnel peut travailler pour plusieurs entreprises (en tant que consultant par exemple) ou non (s'il est indépendant).

Créez une classe **Enterprise** qui comprend le nom de l'entreprise et la localité où se trouve le siège central.
Prévoyez les propriétés et une méthode ToString.

Créez deux entreprises dans la classe Program.

Modifiez la classe ProfessionalContact:

- ajoutez un attribut : une liste d'entreprises dans lesquelles travaille le contact professionnel,
- ajoutez la propriété,
- ajoutez une méthode *EnterpriseAdd* qui ajoute à la liste d'entreprises, les coordonnées d'une entreprise dans laquelle le contact travaille.

Dans la classe Program, vous ajoutez les deux entreprises au premier contact professionnel consultant, une des deux entreprises au deuxième contact professionnel.
Créez dans Program une liste avec les trois contacts professionnels.

Garnissez une variable anonyme une liste de tous les contacts professionnels qui sont indépendants en utilisant le langage LINQ. Faites afficher le nombre d'indépendants.

Garnissez une variable anonyme une liste de tous les contacts professionnels qui sont consultants pour l'entreprise XXX (celle ci-dessus que vous avez ajouté aux deux

professionnels consultants) via une expression Lambda. Faites afficher les coordonnées de cette liste.

Deuxième partie : dynamic

Les langages de programmation peuvent être subdivisés en deux groupes :

- langages statiquement typés : C#, Java;
- langages dynamiquement typés : Python, Ruby, JavaScript. Il n'y a pas de vérification de type lors de la compilation mais uniquement à l'exécution.

En C#, des éléments dynamiques ont été ajoutés pour améliorer l'interopérabilité avec des langages et des environnements dynamiques.
Le mot clé **dynamic** assure des fonctionnalités dynamiques tout en restant statiquement typé. Via ce mot clé, le programmeur demande au compilateur de ne pas vérifier le type de la variable lors de la compilation.

Exemple

```
dynamic d = "Informatique ";
Console.WriteLine(d.GetType());     // affichera System.String
d = 100;
Console.WriteLine(d.GetType());   // affichera System.Int32
```

Types Dynamique, object ou var

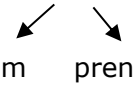
```
Object obj = 10;
Console.WriteLine(obj.GetType());
                // imprime System.Int32 alors que le type statique est System.Object
obj= (int)obj + 10;                          // cast obligatoire
obj= "test";                                  // permis vu System.Object

var varExample = 10;
Console.WriteLine(varExample.GetType());  // imprime System.Int32
varExample += 10;                             // pas de cast
varExample = "test";                          // erreur à la compilation

dynamic dynamicExample = 10;
Console.WriteLine(dynamicExample.GetType());   // imprime System.Int32
dynamicExample += 10;                                // pas de cast
dynamicExample = "test";
                                  // permis! System.Object sous-entendu par dynamic;
```

Etape 1 : ajouter dans la classe PrivatedContact

- méthode Print() qui renvoie la chaîne de caractères *… … est un contact privé*



Etape 2 : ajouter dans la classe ProfessionalContact

- méthode Print() qui renvoie la chaîne de caractères *... ... est un contact professionnel*

Etape 3 : ajouter une classe Car

- attribut : numéro de plaque
- constructeur
- ToString simple avec la valeur du numéro de plaque

Etape 4 : ajouter une classe ContactCar

- attributs :
  - person : référence vers une Personne
  - car : référence vers une voiture
- constructeur général
- méthode *DynamicPrint* qui, recevant un objet dynamique, affiche le return de la méthode Print() appliquée sur l'objet :

```
public void DynamicPrint(dynamic objet)
{
    System.Console.WriteLine(objet.Print() + " voiture : " + Car.ToString());

}
```

Dans le cas où **objet** sera un objet de type PrivateContact, ce sera la chaîne de caractères (nom prénom est un contact privé) ; dans le cas d'un objet de type ProfessionalContact, ce sera l'autre chaîne (nom prénom est un contact professionnel).

Etape 4 : dans main,

Créez un objet de type Car.
Créez un objet de type ContactCar où la personne sera un des contacts de type PrivateContact.
Faites appel à la méthode DynamicPrint appliquée à l'objet de type ContactCar et avec paramètre l'objet de type PrivateContact.

Créez un objet de type Car.
Créez un objet de type ContactCar où la personne sera un des contacts de type ProfessionalContact.
Faites appel à la méthode DynamicPrint appliquée à l'objet de type ContactCar et avec paramètre l'objet de type ProfessionalContact.

Troisième partie : approche des délégués

Vous reprenez votre premier exercice. Vérifiez que vos classes sont de type public.

Le but est de créer une méthode d'affichage **PrintPupilActivityCompulsory** qui permettra d'imprimer:

… âgée(e) de … ans a choisi les activités (obligatoires) :

1. …
2. …
3. …

où la ligne d'impression de l'activité, hormis le fait de la numéroter, sera laissée au choix du programmeur le moment venu. Conceptuellement, la zone encadrée ci-dessus doit travailler avec un objet de type Activity et renvoyer une chaîne de caractères.
Dans la méthode d'affichage **PrintPupilActivityCompulsory**, nous n'encoderons pas l'affichage de la partie de ligne encadrée ci-dessus et la remplacerons par MyPrintActivity.

En premier lieu, il faut déclarer un délégué :

public **delegate** string **DelegatePrintActivityCompulsory** (Activity activity) ;

avant/après les propriétés (pour raison de clarté).
DelegatePrintActivityCompulsory est le nom donné au délégué ; il renvoie une chaîne de caractères à partir d'un objet de type Activity.

Le code de la méthode demandée sera :

```
public string PrintPupilActivityCompulsory (DelegatePrintActivityCompulsory MyPrintActivity)
    {
        int numAct = 0;
        string ch = base.ToString() + " a choisi les activités obligatoires : \n";
        foreach (Activity activity in LstActivities)
            if (activity.Compulsory)
                ch += (++numAct) + " " + MyPrintActivity (activity);
        return ch;
    }
```

où MyPrintActivity sera la méthode choisie par le programmeur.

Il y a plusieurs manières d'implémenter le code qui correspond à MyPrintActivity.
Avant de tester, assurez-vous que vous avez bien un élève qui suit des activités obligatoires (nous n'envisageons pas les cas d'erreurs). Ci-dessous, c'est pupilActComp.

Dans la classe de test, nous allons écrire :

```
System.Console.Write(pupilActComp.PrintPupilActivityCompulsory(…));
```

Où remplacer les pointillés peut se faire de 4 manières différentes (étapes ci-après).

Etape 3 : première manière : implémentation du délégué de manière anonyme

Les pointillés représentent le code de la méthode qui doit remplacer MyPrintActivity.

De manière anonyme, on peut placer le code directement :

```
System.Console.WriteLine (pupilActComp. PrintPupilActivityCompulsory(
        delegate(Activity activity)
        {
           return activity.Title+\n";
        } ) ) ;
```

Etape 4 : deuxième manière : implémentation via une méthode statique dans la classe qui comprend main

Les pointillés représentent l'appel de la méthode statique, méthode à votre choix que vous placez en-dessous de la méthode main dans la classe Program.

Exemple :

```
  private static string  StaticPrintActivity(Activity activity)
 {
        return activity.Title + "\n";
 }
```

Nous obtenons :

```
System.Console.WriteLine(pupilActComp.PrintPupilActivityCompulsory(StaticPrintActivity
)) ;
```

Etape 5 : 3ème manière : implémentation via une autre classe qui comprendra l'affichage

Les pointillés représentent l'appel de la méthode qui doit être appliquée sur un objet, instance de la classe à créer.

Soit la classe **PrintActivityDelegate.** Elle ne comprendra que la méthode :

```
    public class PrintActivityDelegate {
        public string PrintActivity(Activity activity)
       {
            return activity.Title + "\n";
        }
    }
```

Un constructeur par défaut est sous-entendu.

Nous obtenons ainsi dans main :

```
PrintActivityDelegate p = new PrintActivityDelegate () ;
System.Console.WriteLine(pupilActComp.PrintPupilActivityCompulsory(p.PrintActivity)) ;
```

Etape 6 : 4ème manière : utilisons l'expression lambda

```
System.Console.Writeline(pupilActComp.PrintPupilActivityCompulsory(activity =>
activity.Title+ Environment.NewLine) ;
```