Ecole Nationale des Sciences Appliquées de Safi

A.U:2019-2020

Développement en c#.NET

Ing. Meryem OUARRACHI

Plan du module

Langage C#

- L'environnement .Net
- Initiation à la programmation C#
- **Programmation Orienté Objet C#**

Programmation avancée en .Net ,C#

- Programmation distribuée
- Gestion de base de donnée
- Application WPF

CHAPITRE3:

Programmation O.O en C#

Les interfaces

- C’est une classe composée d’un ensemble de méthodes abstraites

interface uneInterface {

nomMeth1 (paramètres) ;

nomMeth2 (paramètres) ;

…

}

- Les classes qui implémentent une interface ,doivent obligatoirement redéfinir toutes les méthodes de cet interface

public class UneClasse : uneInterface

- Une classe peut **simultanément** dériver et implémenter (la dérivation est indiquée avant l’implémentation).

- Une classe peut hériter de plusieurs interfaces.

public class UneClasse : uneInterface1 , uneInterface2

Les interfaces

```
public interface UneInterface {
      void m1();
      void m2();
       }
```

```
public class ClasseInstanciable:UneInterface {
          public void m1() {...}
          public void m2() {...}
       }
```

Les interfaces

- Une interface est un type qui:

-Ne peut pas avoir des constructeurs et des champs.

-Ne spécifie aucun modificateur d'accès pour ses membres, ils sont tous implicitement public et abstract.

-Ne peut pas avoir des membres statiques.

-Ne peut jamais hériter d'une classe même si celle-ci est abstraite.

-Peut hériter d'autres interfaces (une ou plusieurs).

Les interfaces

- Une interface est un type qui:

-Ne peut pas avoir des constructeurs et des champs.

-Ne spécifie aucun modificateur d'accès pour ses membres, ils sont tous implicitement public et abstract.

-Ne peut pas avoir des membres statiques.

-Ne peut jamais hériter d'une classe même si celle-ci est abstraite.

-Peut hériter d'autres interfaces (une ou plusieurs).

Les interfaces: Implémentation explicite

- **Problématique**

```
//2 interfaces avec 2 méthodes identiques en terme de signaure
interface Iinterface1{ void methode1(); }

interface Iinterface2 { void methode1(); }
```

```
class class1: Iinterface1, Iinterface2
{
    public void methode1()
    {// s'agit-il de la méthode de l'interface1 ou bien de Iinterface2?
    }//ici il n'y a pas de distinction, une seule implémentation
}
```

Les interfaces: Implémentation explicite

- **Problématique**

```
Iinterface1  c1 = new class1();
Iinterface2  c2 = new class1();
class1 c3 = new class1();
c1.methode1();//On veut l'exécution de methode1 de interface1 pas ce qu'on aura
c2.methode1();// On veut l'exécution de methode1 de interface1 pas ce qu'on aura
c3.methode1();//Exécution de la méthode implémentée
```

Les interfaces: Implémentation explicite

- **Solution**

```
//2 interfaces avec 2 méthodes identiques en terme de signaure
interface Iinterface1{   void methode1(); }

interface Iinterface2 {  void methode1(); }

class class1: Iinterface1, Iinterface2
{
    void Iinterface1.methode1()
    {//Le mot clé public n'est pas autorisé sur
        //une déclaration d'interface explicite
    }

      void Iinterface2.methode1()
    {//Le mot clé public n'est pas autorisé sur
        //une déclaration d'interface explicite
    }

    public  void methode1()
    {//Le mot clé public n'est pas autorisé sur
        //une déclaration d'interface explicite
    }
}
```

Les interfaces: Implémentation explicite

- **Solution**

```
Iinterface1  c1 = new class1();

Iinterface2  c2 = new class1();

class1 c3 = new class1();

c1.methode1();//On va avoir l'exécution de methode1 de interface1
c2.methode1();//On va avoir l'exécution de methode1 de interface2
c3.methode1();//Exécution de la méthode implémentée
```

Le mot clé « is »

-Ce mot clé « is » permet de vérifier le type d'une variable:

```
if (obj1 is class1)
{
}
```

Les Conversions

```
int x = 7;
long y = x; // Cast implicite

double z = (double)x; //Cast explicite

object  o="salut" ;
string s = o as string;// cast via le mot clé as
```

Remarque:

-as ne prend pas en considération les types values.

-Contrairement au cast explicite, le mot clé as ne déclenche pas une exception si le type n'est pas correct mais retourne un null.

Les collections

Les collections

- Une collection est un tableau dynamique d'objets.
- En C#, on a plusieurs types de collections:
  - -List
  - -ArrayList
  - -Vector
  - -Dictionary
- Une collection fournit un ensemble de méthodes qui permettent:
  - -D'ajouter ou supprimer un objet
  - -Rechercher un élément
  - -Trier les éléments
  - -etc

List

- récupère un élément de la liste:

  List<int> p =new List<int> ()

  p[int index]

- Parmi les méthodes proposées par cette classe on a :

-void Add (Object elt) : insertion de l'élément elt à un index de la liste

-void RemoveAt(int index) : supprime un élément à un index de la liste

-int Count : retourne le nombre d’éléments de la liste

-Object Max(): récupère le maximum d’une liste

-Object[]:Sort():pour trier la liste

List

```
List<int> f = new List<int>();
f.Add(100);
f.Add(200);
f.Add(90);
```

```
//parcourir la liste
for (int i = 0; i < f.Count; i++)
{Console.WriteLine("value N[{0}]={1}" ,i, f[i]); }
```

```
//recupérer le maximum
Console.WriteLine("le maximum dans la liste est"+f.Max());
```

```
//Trier la liste
f.Sort();
for (int i = 0; i < f.Count; i++)
{Console.WriteLine("value N[{0}]={1}", i, f[i]); }
```

Dictionary

- Un dictionnaire est une collection d'objet de type clé/ Valeur. Chaque clé référence l'objet dans le dictionnaire.

```
Dictionary<int, string> dc = new Dictionary<int, string>();
```

dc[int index]: récupère un élément de dictionnaire de la clé index

- **Les méthodes:**

-Object Add(Object key, Object value) : ajoute un élément dans le dictionnaire

- void Clear() : supprime tous les éléments de dictionnaire,

- Values : retourne tous les éléments de la table,

- keys : retourne cette fois ci toutes les clefs,

- int Count() : nombre d ’éléments dans la table,

- Object Remove(Object key) : supprime un élément.

Dictionary

```
Dictionary<int, string> dc = new Dictionary<int, string>();

dc.Add(1, "printemps");
dc.Add(10, "été");
dc.Add(12, "automne");
dc.Add(45, "hiver");
```

```
//récupérer les valeurs
foreach (string v in dc.Values)
{ Console.WriteLine("les valeurs = {0}", v); }
```

```
//récupérer les clés
foreach (int c in dc.Keys)
{ Console.WriteLine("les clés = {0}", c); }
```

Les interfaces IComparable / IComparer

•Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

- IComparable : interface pour définir un ordre de tri naturel pour un objet

- IComparer : interface pour définir un ordre de tri quelconque

Interface IComparable

- Elle propose une méthode :

int CompareTo(Object o)

Elle doit renvoyer :

- <0 si this est **inférieur à o**
- ==0 si this est **égal à o**
- >0 si this est **supérieur à o**

Interface IComparable

```
public class Personne : IComparable

{ public String nom;
    public Personne(String nom)
    { this.nom = nom; }

    public int CompareTo(object o)
    { Personne p = o as Personne;
      return nom.CompareTo(p.nom);
    }   }

class test
{  static void Main(string[] args)
    {  List<Personne> p = new List<Personne>();
        p.Add(new Personne("ouarrachi"));
        p.Add(new Personne("nouri"));

        p.Sort();
        foreach (var v in p)
        { Console.WriteLine(v.nom); }
```

Interface IComparable

```
public class Personne : IComparable<Personne>
{
     public String nom;
     public Personne(String nom)
     { this.nom = nom; }

     public int CompareTo(Personne p)
     {return nom.CompareTo(p.nom);     }          }

class test
{   static void Main(string[] args)
     {   List<Personne> p = new List<Personne>();
          p.Add(new Personne("ouarrachi"));
          p.Add(new Personne("nouri"));

          p.Sort();
          foreach (var v in p)
          { Console.WriteLine(v.nom); }   }  }
```

Interface IComparer

- Elle propose une méthode :

int Compare(Object o1, Object o2)

Elle doit renvoyer :

- Un entier positif si o1 est "plus grand" que o2.

- 0 si la valeur de o1 est identique à celle de o2.

- Une entier négatif si o1 est "plus petit" que o2.

Interface IComparer

```
public class ComparatorPersonne : IComparer<Personne>
{
    public int Compare(Personne p1, Personne p2)
    {
        String nom1 = p1.Nom;
        String nom2 = p2.Nom;
        return nom1.CompareTo(nom2);
    }
}
```

Interface IComparer

```
static void Main(string[] args)
 {  List<Personne> p = new List<Personne>();

     p.Add(new Personne("ouarrachi"));
     p.Add(new Personne("aaaa"));
     p.Add(new Personne("nouri"));

     p.Sort(new ComparatorPersonne());

    Console.WriteLine(p.BinarySearch(new Personne("nouri"), new ComparatorPersonne()));
```

LINQ : Language INtegrated Query

-Linq est une technologie permettant de faciliter la gestion de divers sources de donnée

-Base de donnée relationnelles SQL,

-Fichier XML

-Collections

-Tableau

via l'utilsation d'un seul modèle

Les développeurs ne sont pas obligés d'apprendre le langage de requête pour chaque type de donnée

LINQ : Language INtegrated Query

-Linq propose des requêtes qui s'exécutent de la même façon pour n'importe quel format de données

-Les requêtes Linq utilisent toujours le modèle objet pour interroger les données

Syntaxe d'une requête Linq

List<Personne> per = new List<Personne> ();

Sélectionner des éléments de la collection	**var x = from p in per** **select p;**
Sélectionner des éléments de la collection selon une condition	**var x = from p in per** **where p.age==17** **select c**
Trier les éléments de la collection	**var x= from p in per** **Order by p.age ascending** **select p;**

Syntaxe d’une requête Linq

Parcourir les éléments du résultat de select	foreach (var i in **x**) { console.writeline(i.nom); }

Les Génériques

-Problème: Ecrire une méthode qui permet de trier n'importe quel type de tableau et récupérer ce dernier.

-Solution

```
static int[] tri(int[] tab)
{  Array.Sort(tab);
      return tab;
}
```

```
static double[]
tri(double[] tab)
{  Array.Sort(tab);
      return tab;
}
```

Ecrire une seule méthode qui peut s'adapter avec n'importe quel type→**Généricité**

Les Génériques

-Avec les génériques, on crée des **méthodes ou des classes** qui sont indépendantes d'un type. On les appellera des méthodes génériques et des types génériques.

-Mécanisme qui permet de faire beaucoup de choses, en termes de réutilisabilité et d'amélioration des performances.

Les Génériques

-Méthode générique

```
static T[] tri<T>(T[] tab)
   {//Méthode générique
       Array.Sort(tab);
       return tab;}
```

-Pour l'appeler:

```
{//appliquer sur des doubles
double[] t1 = { 10, 2.4, 15, 8 };
 double[] t2 = tri<double>(t1);
{//appliquer sur des int
 int[] t3 = { 10, 2.4, 15, 8 };
 int[] t4 = tri<int>(t3);
```

Les Génériques

-Méthode générique

-Il est bien sûr possible de créer des méthodes prenant en paramètres plusieurs types génériques différents. Il suffit alors de préciser autant de types différents entre les chevrons qu'il y a de types génériques différents :

```
public static bool EstEgal<T, U>(T t, U u)
{
    return t.Equals(u);
}
```

Les Génériques

-Classe générique:

-Une classe générique fonctionne comme pour les méthodes. C'est une classe où nous pouvons indiquer des types génériques.

-C'est comme cela que fonctionne la classe List. On pourra utiliser cette classe avec tous les types grâce <> .Cela évite de créer une classe ListInt,ListString…

-Syntaxe

```
public class MaClasseGenerique <T>
{
}
```

Les Classes génériques

-Exemple:

```
//Classe Générique
class ClassGénrique <T>
{ //Champs générique
  private T genericField1;
  private T genericField2;
```

```
//Constructeur Générique
 public ClassGénrique(T
genericParam)
 {
this.genericField1 = genericParam;
//Affectation de la valeur par
défaut du type qui remplacera 'T'
this.genericField2 = default(T);
        }
```

```
//Propriété Générique
public T GenericField1
        {
            get { return
genericField1; }
            set {
genericField1 = value; }
        }
```

```
//Méthode Générique
 public T Get() {
 return genericField1; }
```

Les Classes génériques

-Remarque:

-Les méthodes génériques possèdent leurs propres types génériques qui sont indépendants de la classe.

-Elles peuvent être déclarées même dans des classes qui ne sont pas génériques.

```
//Méthode générique dans une classe générique.
class MaClasseGeneric<T> {
    //le type générique U est indépendant du type générique 'T' de la
    //classe et appartient à la méthode Meth.
    void Meth<U>() { }
}

//Méthode générique dans une classe non générique.
class MaClasse {
    void Meth<U>() { }    }
```

Ajouter des contraintes sur les types génériques

-On utilise le mot clé where pour ajouter des contraintes au type générique T.

```
//le type T doit hériter de la classe ClasseX.
class Class1<T> where T : ClasseX {}

//le type T doit hériter de l'interface IInterfaceX.
class Class2<T> where T : IInterfaceX {}

//le type T doit être un Type Reference .
class Class3<T> where T : class {}

//le type T doit être un Type valeur.
class Class4<T> where T : struct {}

//le type T doit être un type concrèt (n'est pas une interface ou un
//type abstrait) qui possède un constructeur par défaut.
class Class5<T> where T : new() {}
```

Ajouter des contraintes sur les types génériques

-On peut combiner les contraintes:

```
//le type T doit être un Reference Type qui hérite de l'interface
//IInterface et qui est un type concrèt avec un constructeur par défaut.
class Class1<T> where T : class, IInterface, new() {}
```

-Avec plusieurs types génériques:

```
class Class1<T, U, V> where T : class, IInterface, new()
                      where U : class, new()
                      where V : new()
{
}
```

Ajouter des contraintes sur les types génériques

- **Quelques utilisations interdites:**

```
//interdit d'avoir une classe et le mot clé class ou struct à la fois
where T : class, ClasseX //Compilation Error
where T : struct, ClasseX //Compilation Error

//interdit d'avoir struct et class à la fois
where T : class, struct //Compilation Error

//interdit d'avoir struct et new() à la fois
where T : struct, new() //Compilation Error

//class et struct doivent être les 1ères contraintes indiquées
where T : IInterface, class //Compilation Error
where T : IInterface, struct //Compilation Error

//new() doit être la dernière contrainte indiquée
where T : new(), IInterface //Compilation Error

//Les classes doivent précéder les interfaces.
where T : IInterface, ClasseX //Compilation Error
```