

# C# : notions avancés

## Les classes partielles

- offrent la possibilité de définir une classe en plusieurs fois.
  - Si la classe devient très longue, elle pourra être découpée en plusieurs fichiers pour regrouper des fonctionnalités qui se ressemblent.
- On utilise pour cela le mot-clé **partial**.
- Utile si une partie de l'implémentation est générée ailleurs.

# C# : notions avancés

## Les classes partielles

- Exp :

```
public partial class Voiture Source1.cs
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }
}
```

```
public partial class Voiture Source2.cs
{
    public string Rouler()
    {
        return "Je roule à " + Vitesse + " km/h";
    }
}
```

# C# : notions avancés

## Classes statiques et méthodes statiques

- le mot-clé **static** permet d'indiquer que la méthode d'une classe n'appartient pas à une instance de la classe.
  - **static** permet d'avoir accès à cette méthode en dehors de tout objet.
- la méthode `Main()` est obligatoirement statique.
  - Le CLR n'a pas besoin d'instancier la classe `Program` pour démarrer notre application. Il a juste à appeler la méthode `Program.Main()`.

# C# : notions avancés

## Classes statiques et méthodes statiques

- Si une classe ne contient que des choses statiques alors elle peut devenir également statique.
  - Cette classe deviendra non-instanciable
- les classes statiques servent à regrouper des méthodes utilitaires
- Une méthode statique ne travaille qu'avec les membres non statiques de sa propre classe.

# C# : notions avancées

## Classes statiques et méthodes statiques

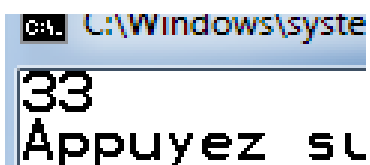
- Exp :

```
public static class Math
{
    public static int Addition(int a, int b)
    {
        return a + b;
    }
}
```

```
int a = 12;
```

```
int b = 21;
```

```
int somme = Math.Addition(a, b);
Console.WriteLine(somme);
```



C:\Windows\system32  
33 Appuyez sur

# C# : notions avancées

## Classes statiques et méthodes statiques

- Attention: 

```
public class Math
{
    public static int Addition(int a, int b)
    {
        return a + b;
    }
}
```

```
int a = 12;
int b = 21;
```

```
Math m = new Math();
int somme = m.Addition(a, b);
Console.Writ
```

(variable locale) int b

Erreur :

Le membre 'cours.Math.Addition(int, int)' est inaccessible avec une référence d'instance ; qualifiez-le avec un nom de type

# C# : notions avancés

## Les classes internes

- Les classes internes (*nested class*) sont un mécanisme qui permet d'avoir des classes définies à l'intérieur d'autres classes.
- Cela peut être utile si on souhaite restreindre l'accès d'une classe uniquement à sa classe mère.
- Cela permet éventuellement de regrouper les classes de manière sémantique
  - l'intérêt d'utiliser une classe interne est moindre, c'est plutôt le but des espaces de noms.

# C# : notions avancées

## Les classes internes

- Exp :

```
class Eleve
{
    public string nom { get; set; }
    public int numIns { get; set; }

    private class Note {
        List<int> matières;

        public double calculerMoyenne()
        {
            //...
        }
    }

    public Eleve()
```



# C# : notions avancés

## Les génériques

- Les génériques sont une fonctionnalité du framework .NET apparus avec la version 2.
- Ils permettent de créer des méthodes ou des classes qui sont indépendantes d'un type.
  - très intéressant en termes de réutilisabilité et d'amélioration des performances.
- Exp : List → List<int>, List<Eleve> , ...

# C# : notions avancées

## Les génériques

- Exp :

```
public static class Afficheur
{
    public static void Affiche(object o)
    {
        Console.WriteLine("Afficheur d'objet :");
        Console.WriteLine("\tType : " + o.GetType());
        Console.WriteLine("\tReprésentation : " + o.ToString());
    }
}
```

```
int i = 5;
double d = 9.5;
string s = "abcd";
Eleve ee = new Eleve();
Afficheur.Affiche(i);
Afficheur.Affiche(d);
Afficheur.Affiche(s);
Afficheur.Affiche(ee);
```

```
Afficheur d'objet :
      Type : System.Int32
      Représentation : 5
Afficheur d'objet :
      Type : System.Double
      Représentation : 9,5
Afficheur d'objet :
      Type : System.String
      Représentation : abcd
```

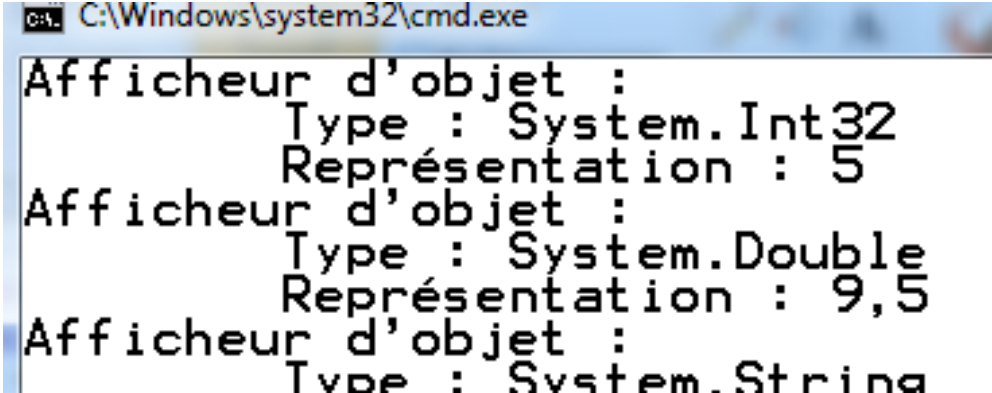
# C# : notions avancées

## Les génériques

- Exp :

```
public static class Afficheur
{
    public static void Affiche<T>(T a)
    {
        Console.WriteLine("Afficheur d'objet :");
        Console.WriteLine("\tType : " + a.GetType());
        Console.WriteLine("\tReprésentation : " + a.ToString());
    }
}
```

```
int i = 5;
double d = 9.5;
string s = "abcd";
Eleve ee = new Eleve();
Afficheur.Affiche<int>(i);
Afficheur.Affiche<double>(d);
Afficheur.Affiche<string>(s);
Afficheur.Affiche<Eleve>(ee);
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of the C# program. It shows three lines of output, each corresponding to one of the calls to the Afficheur.Affiche method. The first line shows the output for an integer (5), the second for a double (9.5), and the third for a string ("abcd"). The output is formatted with tabs to align the Type and Représentation fields.

```
Afficheur d'objet :
                Type : System.Int32
                Représentation : 5
Afficheur d'objet :
                Type : System.Double
                Représentation : 9,5
Afficheur d'objet :
                Type : System.String
```

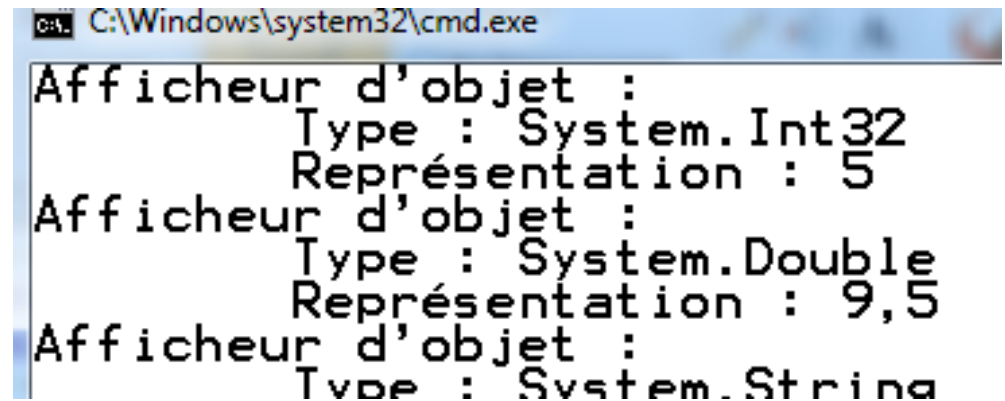
# C# : notions avancées

## Les génériques

- Exp :

```
public static class Afficheur
{
    public static void Affiche<T>(T a)
    {
        Console.WriteLine("Afficheur d'objet :");
        Console.WriteLine("\tType : " + a.GetType());
        Console.WriteLine("\tReprésentation : " + a.ToString());
    }
}
```

```
int i = 5;
double d = 9.5;
string s = "abcd";
Eleve ee = new Eleve();
Afficheur.Affiche(i);
Afficheur.Affiche(d);
Afficheur.Affiche(s);
Afficheur.Affiche(ee);
```



```
C:\Windows\system32\cmd.exe
Afficheur d'objet :
        Type : System.Int32
        Représentation : 5
Afficheur d'objet :
        Type : System.Double
        Représentation : 9,5
Afficheur d'objet :
        Type : System.String
```

**Le compilateur devine le type**

# C# : notions avancées

## Les génériques

- Exp :
  - Avec la version objet c'est le boxing qui est utilisé
  - Avec l'aspect générique on améliore les performances (on évite le boxing / unboxing)
  - Le type générique est T par convention
  - On peut utiliser des paramètres de type générique différents

```
public bool comparer<T, U>(T a, U b) {  
    return a.Equals(b);  
}
```

# C# : notions avancées

## Les génériques

- Exp :

classe générique

– On a

Plusieurs type

Class C <T, U, ...>

```
public class MaListeGenerique<T>
{
    private int capacite;
    private int nbElements;
    private T[] tableau;

    public MaListeGenerique()
    {
        capacite = 10;
        nbElements = 0;
        tableau = new T[capacite];
    }

    public T ObtenirElement(int indice)
    {
        return tableau[indice];
    }

    public void Ajouter(T element)
    {
        //...
    }
}
```

# C# : notions avancés

## Les génériques

- Exp :
- La méthode obtenirElement
  - Pour un objet non initié retournera **null**.
  - Mais pour un type valeur elle n'aura pas de sens.
- le mot-clé **default** permet de renvoyer la valeur par défaut du type.

```
public T ObtenirElement(int indice)
{
    if (indice < 0 || indice >= nbElements)
    {
        Console.WriteLine("L'indice n'est pas bon");
        return default(T);
    }
    return tableau[indice];
}
```

# C# : notions avancées

## Les génériques

- Les interfaces peuvent aussi être génériques.

```
public class Voiture : IComparable<Voiture>
{
    public string Couleur { get; set; }
    public string Marque { get; set; }
    public int Vitesse { get; set; }
    public int CompareTo(Voiture obj)
    {
        return Vitesse.CompareTo(obj.Vitesse);
    }
}
```

- Plus besoin de catser le paramètre



# C# : notions avancés

## Les génériques

- Il existe 6 types de restrictions :

Contrainte	Description
where T : struct	Le type générique doit être un type valeur
where T : class	Le type générique doit être un type référence
where T : new()	Le type générique doit posséder un constructeur par défaut
where T : IMonInterface	Le type générique doit implémenter l'interface IMonInterface
where T : MaClasse	Le type générique doit dériver de la classe MaClasse
where T1 : T2	Le type générique doit dériver du type générique T2


```
public class MyClass<T> where T : struct  
{
```

# C# : notions avancées

## Les types nullable

- Il s'agit de permettre à un type valeur d'avoir une valeur nulle.
  - Et ce via la classe Nullable<>.

```
Nullable<int> entier = null; //...
if (!entier.HasValue)
{
    Console.WriteLine("l'entier n'a pas de valeur");
}
else
{
    Console.WriteLine("Valeur de l'entier : " + entier.Value);
}
```



A blue arrow points from the `Nullable<int>` type in the first line of the code to a box containing the shorthand `int?`.

```
int? entier = null;
//...
```

# Les collections

- des classes spécialisées dans le stockage et la récupération des données.
- Se sont de bonnes alternatives aux tableaux
- Deux catégories de collections
  - Standards : `System.Collections`
  - Génériques : `System.Collections.Generic`
    - Cette deuxième catégorie est plus intéressante d'ailleurs elle est chargée par défaut par Visual C#

# Les collections : ArrayList

- Collection standard du namespace System.Collections
- C'est un tableau dynamiquement extensible
- Hétérogène : Peut contenir des éléments de types différents

```
ArrayList da = new ArrayList();
```

```
da.Add("C Sharp");  
da.Add(344);  
da.Add(55);  
da.Add(new Eleve());  
da.Remove(55);
```

```
foreach (object el in da)  
    Console.WriteLine(el);
```

# Les collections : List

- Générique : namespace System.Collections.Generic

```
List<string> langs = new List<string>();
```

```
langs.Add("Java");  
langs.Add("C#");  
langs.Add("C");  
langs.Add("C++");
```

```
Console.WriteLine(langs.Contains("C#"));
```

```
Console.WriteLine(langs[1]);  
Console.WriteLine(langs[2]);
```

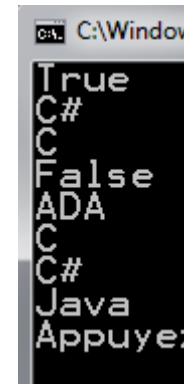
```
langs.Remove("C++");
```

```
Console.WriteLine(langs.Contains("C++"));
```

```
langs.Insert(2, "ADA");
```

```
langs.Sort();
```

```
foreach (string lang in langs)  
    Console.WriteLine(lang);
```



# Les collections : Dictionary

- Générique. Appelé aussi tableau associatif :
  - Gère des clés unique auxquels sont associées des valeurs

```
Dictionary<string, string> domains = new Dictionary<string, string>();
```

```
domains.Add("de", "Germany");  
domains.Add("sk", "Slovakia");  
domains.Add("us", "United States");  
domains.Add("ru", "Russia");  
domains.Add("hu", "Hungary");  
domains.Add("pl", "Poland");
```

```
Console.WriteLine(domains["sk"]);  
Console.WriteLine(domains["de"]);
```

```
Console.WriteLine("Dictionary has {0} items", domains.Count);
```

# Les collections : Dictionary

```
Console.WriteLine("Keys of the dictionary:");

List<string> keys = new List<string>(domains.Keys);

foreach (string key in keys)
    Console.WriteLine("{0}", key);

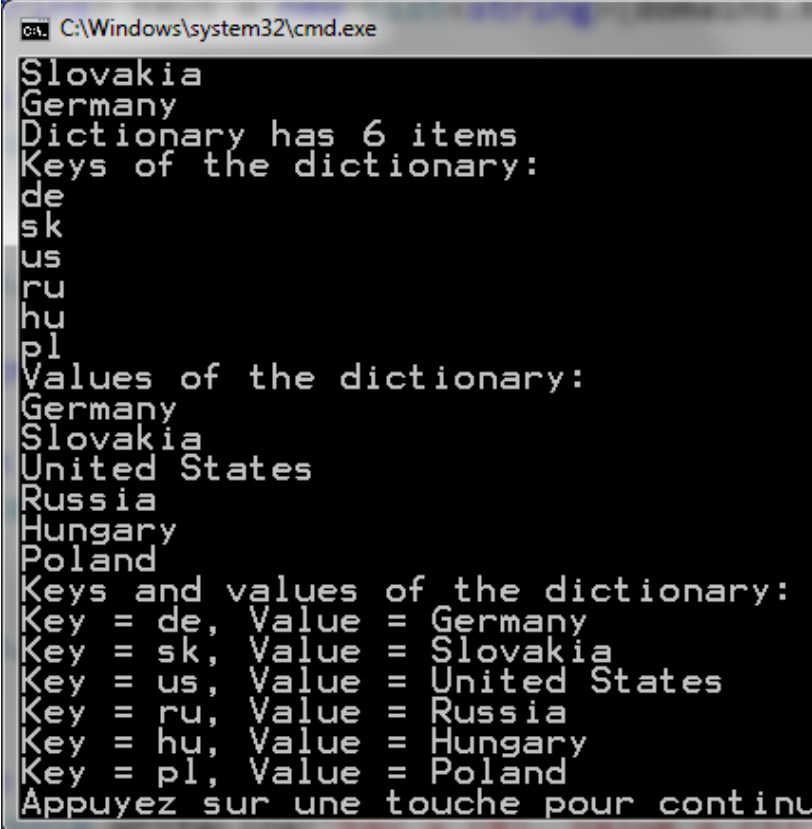
Console.WriteLine("Values of the dictionary:");

List<string> vals = new List<string>(domains.Values);

foreach (string val in vals)
    Console.WriteLine("{0}", val);

Console.WriteLine("Keys and values of the dictionary:");

foreach (KeyValuePair<string, string> kvp in domains)
    Console.WriteLine("Key = {0}, Value = {1}",
        kvp.Key, kvp.Value);
```



```
C:\Windows\system32\cmd.exe
Slovakia
Germany
Dictionary has 6 items
Keys of the dictionary:
de
sk
us
ru
hu
pl
Values of the dictionary:
Germany
Slovakia
United States
Russia
Hungary
Poland
Keys and values of the dictionary:
Key = de, Value = Germany
Key = sk, Value = Slovakia
Key = us, Value = United States
Key = ru, Value = Russia
Key = hu, Value = Hungary
Key = pl, Value = Poland
Appuyez sur une touche pour continuer
```

# Les collections : Queue

- C'est une file : First-In-First-Out (FIFO)

```
Queue<string> msgs = new Queue<string>();
```

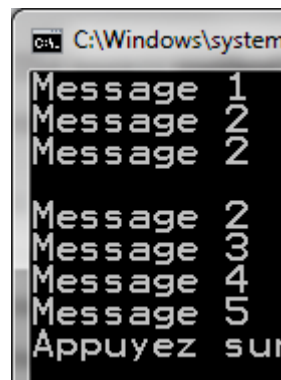
```
msgs.Enqueue("Message 1");  
msgs.Enqueue("Message 2");  
msgs.Enqueue("Message 3");  
msgs.Enqueue("Message 4");  
msgs.Enqueue("Message 5");
```

```
Console.WriteLine(msgs.Dequeue());  
Console.WriteLine(msgs.Peek());  
Console.WriteLine(msgs.Peek());
```

```
Console.WriteLine();
```

```
foreach (string msg in msgs)  
    Console.WriteLine(msg);
```

- Enqueue : ajoute à la fin de la file
- Dequeue : retourne puis supprime la tête de file
- Peek : retourne la tête de la file sans la supprimer





# Les collections : Stack

- C'est une Pile : Last-In-First-Out (LIFO)

```
Stack<int> stc = new Stack<int>();
```

```
stc.Push(1);  
stc.Push(2);  
stc.Push(3);  
stc.Push(4);  
stc.Push(5);
```

```
Console.WriteLine(stc.Pop());  
Console.WriteLine(stc.Peek());  
Console.WriteLine(stc.Peek());
```

```
Console.WriteLine();
```

```
foreach (int item in stc)  
    Console.WriteLine(item);
```



- Push : ajoute en haut de la pile
- Pop : retourne et supprime le haut de la pile
- Peek : retourne le haut de la pile sans le supprimer

# C# : notions avancés

## Les méthodes d'extension

- En général, pour ajouter des fonctionnalités à une classe, on peut
  - soit modifier le code source de la classe,
  - soit créer une classe dérivée et ajouter ces fonctionnalités
- Il existe un autre moyen pour étendre une classe :
  - les méthodes d'extension.
- Elles sont intéressantes si on n'a pas la main sur le code source de la classe ou si la classe n'est pas dérivable (sealed).

```
public sealed class nonDerivable  
{
```

# C# : notions avancés

## Les méthodes d'extension

Exp

- Sachant que String n'est pas dérivable
- Si on souhaite crypter une chaîne de caractères dans un format personnel
  - On peut créer une méthode statique utilitaire faisant cet encodage
  - Ou utiliser une méthode d'extension

# C# : notions avancés

## Les méthodes d'extension

Exp

- Pour créer cette méthode d'extension, Il suffit de mettre **this** devant le paramètre (string) de la méthode.
- Désormais cette méthode pourra être utilisée comme s'il faisait partie de la classe String

# C# : notions avancés

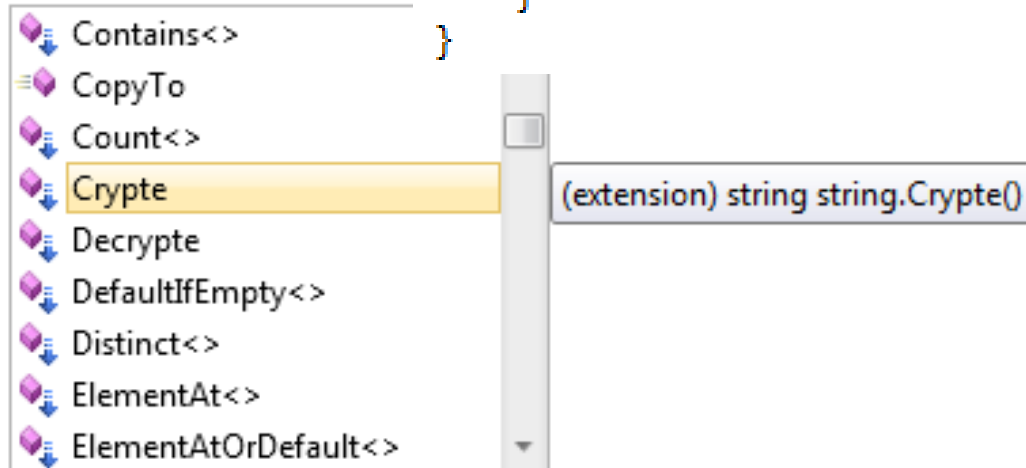
## Les méthodes d'extension

Exp

```
public static class Encodage
{
    public static string Crypte(this string chaine)
    {
        return
            Convert.ToString(Encoding.Default.GetBytes(chaine));
    }
    public static string Decrypte(this string chaine)
    {
        return
            Encoding.Default.GetString(Convert.FromBase64String(chaine));
    }
}
```

```
string s = "Test";
```

s.



# C# : notions avancés

## Les méthodes d'extension

- une méthode d'extension n'a pas accès aux méthodes privées ou variables membres internes à la classe.
  - Ces méthodes doivent donc être statiques et situées à l'intérieur d'une classe statique.
- il faut faire attention à l'espace de nom où se situent nos méthodes d'extensions.
  - Si le **using** correspondant n'est pas inclus, on ne verra pas les méthodes d'extension.
- les méthodes d'extension fonctionnent aussi avec les interfaces.
  - elles étendent toutes les classes qui implémentent une interface.

# C# : notions avancés

## Les délégués (delegate)

- Les délégués sont des variables qui « pointent » vers une méthode.
- Un délégué permettra de définir une signature de méthode, ainsi on pourra pointer avec sur n'importe quelle méthode qui respecte cette signature.
- On utilise un délégué quand on veut passer une méthode en paramètre à une autre méthode.

# C# : notions avancées

## Les délégués (delegate)

- Exp

```
public class TrieurDeTableau
{
    public delegate void DelegateTri(int[] tableau);

    public void TriAscendant(int[] tableau)
    {
        Array.Sort(tableau);
    }

    public void TriDescendant(int[] tableau)
    {
        Array.Sort(tableau);
        Array.Reverse(tableau);
    }
}
```



# C# : notions avancées

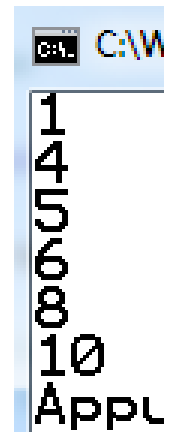
## Les délégués (delegate)

- Exp

```
public void TrierEtAfficher(int[] tableau, DelegateTri methodeDeTri)
{
    methodeDeTri(tableau);
    foreach (int i in tableau)
    {
        Console.WriteLine(i);
    }
}
```

```
static void Main(string[] args)
{
    int[] tableau = new int[] { 4, 1, 6, 10, 8, 5 };

    TrieurDeTableau.DelegateTri tri = new TrieurDeTableau().TriAscendant;
    new TrieurDeTableau().TrierEtAfficher(tableau, tri);
}
```



C:\W  
1  
4  
5  
6  
8  
10  
Appl

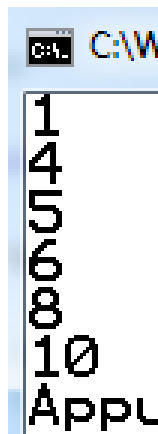
# C# : notions avancées

## Les délégués (delegate)

- Exp : avec méthode anonyme

```
static void Main(string[] args)
{
    int[] tableau = new int[] { 4, 1, 6, 10, 8, 5 };

    new TrieurDeTableau().TrierEtAfficher(tableau,
                                           delegate(int[] leTableau)
                                           {
                                               Array.Sort(leTableau);
                                           }
                                           );
}
```



# C# : notions avancés

## Les délégués Multicast (diffusion multiple)

- Un délégué peut être **multicast**, c'est à dire qu'il peut pointer vers plusieurs méthodes.
- Chaque délégué possède une **liste d'invocation** référençant toutes les méthodes à invoquer quand le délégué est invoqué.
- Il faut signaler que l'ordre d'invocation des méthodes n'est pas garanti et ne dépend pas forcément de l'ordre dans la liste.

# C# : notions avancées

## Les délégués Multicast (diffusion multiple)

- Exp

```
public class TrieurDeTableau
{
    public static void TriAscendant(int[] tableau)
    {
        Array.Sort(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine();
    }

    public static void TriDescendant(int[] tableau)
    {
        Array.Sort(tableau);
        Array.Reverse(tableau);
        foreach (int i in tableau)
        {
            Console.WriteLine(i);
        }
        Console.WriteLine();
    }
}
```

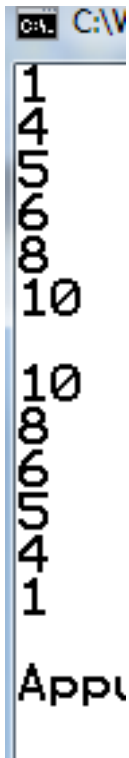
# C# : notions avancées

## Les délégués Multicast (diffusion multiple)

- Exp

```
class Program
{
    public delegate void DelegateTri(int[] tableau);
    static void Main(string[] args)
    {
        int[] tableau = new int[] { 4, 1, 6, 10, 8, 5 };

        DelegateTri tri = TrieurDeTableau.TriAscendant;
        tri += TrieurDeTableau.TriDescendant;
        tri(tableau);
    }
}
```



C:\V

1  
4  
5  
6  
8  
10  
10  
8  
6  
5  
4  
1

Appl

# C# : notions avancés

## Les délégués Multicast (diffusion multiple)

- La classe Delegate possède deux méthodes statiques: **Combine** et **Remove** qui permettent de concaténer ou dissocier les liste d'invocation de deux délégués
- Les opérateurs arithmétiques + et – peuvent remplacer Combine et Remove
- On dispose aussi de += et -=

# C# : notions avancées

## Les délégués Multicast (diffusion multiple)

- Exp `class Program`

```
{  
    public delegate void DelegateTri(int[] tableau);  
    static void Main(string[] args)  
    {  
        int[] tableau = new int[] { 4, 1, 6 };  
  
        DelegateTri tri1 = TrieurDeTableau.TriAscendant;  
        tri1 += TrieurDeTableau.TriDescendant;  
  
        DelegateTri tri2 = TrieurDeTableau.TriAscendant;  
  
        DelegateTri tri = (DelegateTri)Delegate.Combine(tri1, tri2);  
        tri(tableau);  
        tri = tri1 - tri2;  
        tri(tableau);  
    }  
}
```

C:\W

1  
4  
6

6  
4  
1

1  
4  
6

6  
4  
1

Appl

# C# : notions avancés

## Les délégués génériques Action et Func

- `Action<Types >` : est équivalent à créer un délégué qui ne renvoie rien et qui prend une liste de paramètres.
  - exemple : `Action<int[], string>`
- `Func<Types>` : ici, c'est le dernier paramètre générique qui sera du type de retour
  - exemple : `Func<int, int, double>` prend deux entiers en paramètres et renvoie un double.



# C# : notions avancées

## Les expressions lambdas

- il s'agit d'une façon simplifiée d'écrire les délégués
- Exp : ces deux écritures sont équivalents

```
DelegateTri tri = delegate(int[] leTableau)
{
    Array.Sort(leTableau);
};
```

```
DelegateTri tri = (leTableau) =>
{
    Array.Sort(leTableau);
};
```

# C# : notions avancées

## Les expressions lambdas

- Exp : encore ces deux écritures sont équivalents

```
TrierEtAfficher(tableau,  
    delegate(int[] leTableau)  
    {  
        Array.Sort(leTableau);  
    }  
);
```

```
TrierEtAfficher(tableau, leTableau => Array.Sort(leTableau));
```

# C# : notions avancés

## Les événements

- Les événements sont un mécanisme du C# permettant à une classe d'être notifiée d'un changement.
  - Par exemple, on peut vouloir s'abonner à un changement de prix d'une voiture.
- La base des événements est le délégué.
  - On pourra stocker dans un événement un ou plusieurs délégués qui pointent vers des méthodes respectant la signature de l'événement.
- Un événement est défini grâce au mot-clé **event**.

# C# : notions avancées

## Les événements

- Exp :

```
public class Voiture
{
    public delegate void prixChange(decimal
nouveauPrix);
    public event prixChange pc;
    public decimal Prix { get; set; }
    public void Promo()
    {
        Prix = Prix / 2;
        if (pc != null)
            pc(Prix);
    }
}
```



pc

prixChange Voiture.pc

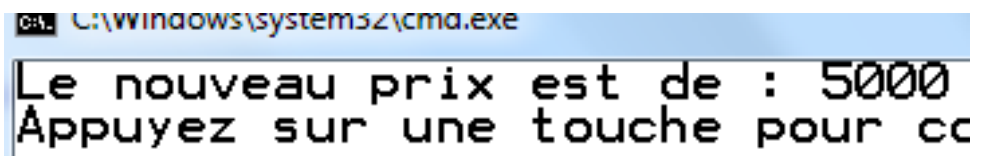
# C# : notions avancées

## Les événements

- Exp :

```
public class DemoEvenement
{
    public void Demo()
    {
        Voiture voiture = new Voiture { Prix = 10000 };
        Voiture.prixChange dpc = changement;
        voiture.pc += dpc;
        voiture.promo();
    }
    private void changement(decimal nouveauPrix)
    {
        Console.WriteLine("Le nouveau prix est de : " +
            nouveauPrix);
    }
}

static void Main(string[] args)
{
    new DemoEvenement().Demo();
}
```



C:\windows\system32\cmd.exe  
Le nouveau prix est de : 5000  
Appuyez sur une touche pour cc