Livret – 3.4

Le langage C# dans .Net

---

generics,

types partiels, méthodes anonymes.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

Les éléments principaux de laVersion 2.0

Les Generics

Plan général:

1. Les generics

1. Les generics ou types paramétrés

Les generics introduits depuis la version 2.0 du C# permettent de construire des classes des structs, des interfaces, des delegates ou des méthodes qui sont paramétrés par un type qu'ils sont capables de stocker et de manipuler.

Le compilateur C# reconnaît un type paramétré (generic) lorsqu'il trouve un ou plusieurs identificateurs de type entre crochets <…> :

Exemples de syntaxe de classes génériques

```
//les types paramétrés peuvent être appliqués aux classes aux interfaces
interface IGeneric1<T>
{
}
class ClassGeneric1< UnType, Autre >
{
}
class ClassInt1 : ClassGeneric1< int, int >
{
}
class ClassInt2 <T> : ClassGeneric1< int, T >
{
}
class ClassInt3 <T, U> : ClassGeneric1< int, U >
{
}
```

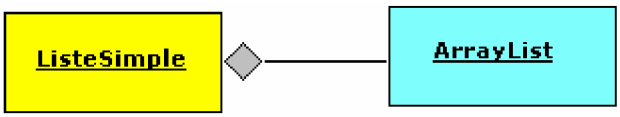
Exemples de syntaxe de méthodes génériques

```
//les types paramétrés peuvent être appliqués aux méthodes
class clA
{
   public void methode1<T>()
   {
   }
   public T[] methode2<T>()
   {
      return new T[10];
   }
}
```

1.1 Une liste générale sans les generics

On pourrait penser que le type object qui est la classe mère des types références et des types valeurs devrait suffire à passer en paramètre n'importe quel type d'objet. Ceci est en effet possible mais avec des risques de mauvais transtypage dont les tests sont à la charge du développeur.

Considérons une classe ListeSimple contenant un objet ArrayList :



Nous souhaitons stocker dans cette classe des entiers de type **int** et afficher un élément de la liste de rang fixé. Nous écrirons un programme dans lequel nous rangeons deux entiers dans un objet listeSimpleInt de type ListeSimple, puis nous les affichons :

```
public class ListeSimple
{
   public ArrayList liste = new ArrayList();

   public void ajouter(object elt)
   {
        this.liste.Add(elt);
   }
   public int afficherInt(int rang)
   {
        return (int)this.liste[rang];
   }
}

class Program
{
   static void Main(string[] args)
   {
      ListeSimple listeSimpleInt = new ListeSimple();
      listeSimpleInt.ajouter(32);
      listeSimpleInt.ajouter(-58);
      int x;
      for(int i=0; i<=1;i++)
      {
         x = listeSimpleInt. afficherInt (i);
         Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
   }
}
```

Qui affichera lors de l'exécution à la console :
listeSimpleInt : 32
listeSimpleInt : -58

Remarque n°1 :

> Dans la méthode afficherInt() l'instruction "**return** (**int**)**this**.liste[rang];". Nous sommes obligés de transtyper l'élément de rang "i" car dans l'ArrayList nous avons stocké des éléments de type object.

Remarque n°2 :

> Nous pouvons parfaitement ranger dans le même objet listeSimpleInt de type ListeSimple des **string** en construisant la méthode afficherStr() de la même manière que la méthode afficherInt( ) :
>
> ```
> public string afficherStr(int rang)
> ```

```
    {
        return (string)this.liste[rang];
    }
```

Soit le code de la classe ainsi construite :

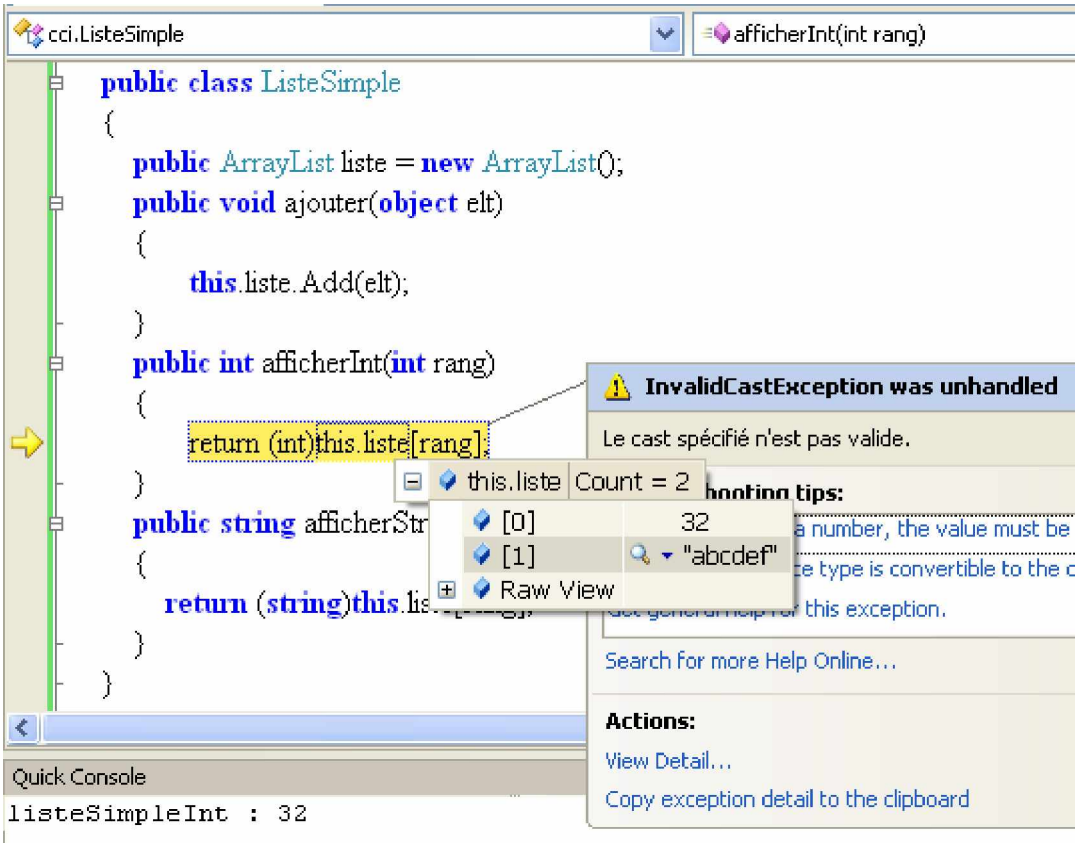
```
public class ListeSimple
{
    public ArrayList liste = new ArrayList();

    public void ajouter(object elt)
    {
        this.liste.Add(elt);
    }
    public int afficherInt(int rang)
    {
        return (int)this.liste[rang];
    }
    public string afficherStr(int rang)
    {
        return (string)this.liste[rang];
    }
}
```

Si nous compilons cette classe, le compilateur n'y verra aucune erreur, car le type **string** hérite du type object tout comme le type **int**. Mais alors, si par malheur nous oublions lors de l'écriture du code d'utilisation de la classe, de ne ranger que des éléments du même type (soit uniquement des **int**, soit uniquement des **string**) alors l'un des transtypage produira une erreur.

Le programme ci-dessous range dans l'ArrayList un premier élément de type **int**, puis un second élément de type **string**, et demande l'affichage de ces deux éléments par la méthode afficherInt( ). Aucune erreur n'est signalée à la compilation alors qu'il y a incompatibilité entre les types. Ceci est normal puisque le type de l'objet est obtenu dynamiquement :

```
class Program
{
    static void Main(string[] args)
    {
        ListeSimple listeSimpleInt = new ListeSimple();
        listeSimpleInt.ajouter ( 32 );
        listeSimpleInt.ajouter ( "abcdef" );
        int x;
        for(int i=0; i<=1;i++)
        {
            x = listeSimpleInt. afficherInt (i);
            Console.WriteLine("listeSimpleInt : " + x);
        }
        Console.WriteLine();
    }
}
```

Le transtypage du deuxième élément qui est une string en un int, est une erreur qui est signalée par le CLR lors de l'exécution :



Il revient donc au développeur dans cette classe à prêter une attention de tous les instants sur les types dynamiques des objets lors de l'exécution et éventuellement de gérer les incompatibilités par des gestionnaires d'exception try…catch.

Nous allons voir dans le prochain paragraphe comment les generics améliorent la programmation de ce problème.

1.2 Une liste générale avec les generics

Nous construisons une classe de type T générique agrégeant une liste d'éléments de type T générique. Dans ce contexte il n'est plus nécessaire de transtyper, ni de construire autant de méthodes que de types différents à stocker.

```
public class ListeGenerique <T>
 {
    List<T> liste = new List<T>();
    public void ajouter ( T elt )
    {
       this.liste.Add(elt);
    }
    public T afficher(int rang)
    {
       return this.liste[rang];
    }
 }
```

C'est dans le code source et donc lors de la compilation que le développeur définit le type <T> de l'élément et c'est le compilateur qui vérifie que chaque élément ajouté est bien de type <T> ainsi que chaque élément affiché est de type <T> :

L'instruction qui suit permet d'instancier à partir de la classe ListeGenerique un objet listeGenricInt d'éléments de type **int** par exemple :

```
ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
```

On peut instancier à partir de la même classe ListeGenerique un autre objet listeGenricStr d'éléments de type **string** par exemple :

```
ListeGenerique< string > listeGenricStr = new ListeGenerique< string >();
```

Le compilateur refusera le mélange des types :

```
class Program
  {
    static void Main(string[] args)
    {
      ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
      listeGenricInt.ajouter ( 32 );
      listeGenricInt.ajouter ( "abcdef" );  ç  erreur signalée par le compilateur ici !
      int x;
      for(int i=0; i<=1;i++)
      {
        x = listeGenricInt.afficher (i);
        Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
    }
  }
```

Un programme correct serait :

```
class Program
  {
    static void Main(string[] args)
    {
      ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
      listeGenricInt.ajouter ( 32 );
      listeGenricInt.ajouter ( -58 );
      int x;
      for(int i=0; i<=1;i++)
      {
        x = listeGenricInt.afficher (i);
        Console.WriteLine("listeSimpleInt : " + x);
      }
      Console.WriteLine();
    }
  }
```

On procéderait d'une manière identique avec un objet "ListeGenerique< **string** > listeGenricStr = **new** ListeGenerique< **string** >()" dans lequel le compilateur n'acceptera que l'ajout d'éléments de type **string**.

1.3 Une méthode avec un paramètre générique

Nous reprenons la construction précédente d'une liste générique :

```
public class ListeGenerique <T>
{
   public List<T> liste = new List<T>();
   public void ajouter ( T elt )
   {
      this.liste.Add(elt);
   }
   public T afficher(int rang)
   {
      return this.liste[rang];
   }
}
```

Nous ajoutons dans la classe Program une méthode générique "**static void** afficher<T>(ListeGenerique<T> objet, **int** rang)" qui reçoit en paramètre formel une liste générique nommée objet de type <T>, nous la chargeons d'afficher l'élément de rang k de la liste et le type dynamique de l'objet obtenu par la méthode GetType(). L'appel de la méthode afficher nécessite deux informations :

a) le type paramétré de définition de l'objet passé en paramètre effectif, soit ici < **int** > ,
b) le paramètre effectif lui-même soit ici listeGenricInt.

```
class Program
   {
      static void afficher<T>(ListeGenerique<T> objet, int k)
      {
         Console.WriteLine(objet.liste[k] + ", de type : " + objet.liste[k].GetType());
      }
       static void Main(string[] args)
      {
         ListeGenerique<int> listeGenricInt = new ListeGenerique<int>();
         listeGenricInt.ajouter ( 32 );
         listeGenricInt.ajouter ( -58 );
         afficher <int> ( listeGenricInt, 0 );
         Console.WriteLine();
      }
   }
```

Résultat obtenu lors de l'exécution sur la console :
32, de type : System.Int32

1.4 Type générique contraint

Il est possible de contraindre un type paramétré à hériter d'une ou plusieurs classes génériques ou non et à implémenter une ou plusieurs interfaces classes génériques ou non en utilisant le mot clef where. Cette contrainte apporte deux avantages au développeur :

1°) améliorer la sécurité de vérification du typage lors de la compilation,
2°) réduire le transtypage.

Les options de contraintes d'un type paramétré sont les suivantes :

Syntaxe de la contrainte	Signification de la contrainte
where T: **struct**	Le type T doit être un type valeur.
where T : **class**	Le type T doit être un type référence.
where T : **new**()	Le type T doit avoir un constructeur sans paramètre explicite.
where T : < classeType>	Le type T doit hériter de la classe classeType
where T : <interfaceType>	Le type T doit implémenter l'interface interfaceType

Le(s) mot(s) clef **where** doit se situer avant le corps de la classe ou de la méthode.

Syntaxe d'utilisation par l'exemple dans une classe :

```
class clA {   ….  }
interface IGeneric1<T>  {   ….  }

class ClassGeneric<UnType, Autre>
    where UnType : clA, new()
    where Autre : class, IGeneric<UnType>
 {
    ……
 }
```

Dans l'exemple précédent la classe ClassGeneric est paramétrée par les deux types UnType et Autre qui supportent chacun une série de contraintes :

- Le type UnType est contraint d'hériter de la classe clA et doit avoir un constructeur sans paramètre explicite.
- Le type Autre est contraint d'être un type référence et doit implémenter l'interface IGeneric avec comme type paramétré <UnType>.

Syntaxe d'utilisation par l'exemple dans une méthode :

```
 public T meth1 < T, U >( )
        where T : new( )
        where U : ClassGeneric <T>
 {
        return new T( );
 }
```

Dans l'exemple précédent la méthode meth1 est paramétrée par les deux types T et U qui supportent chacun une série de contraintes (elle renvoie un résultat de type T) :

- Le type T est contraint d'avoir un constructeur sans paramètre explicite.
- Le type U est contraint d'hériter de la classe ClassGeneric avec <T> comme type paramétré.

1.5 Surcharge générique d'une classe

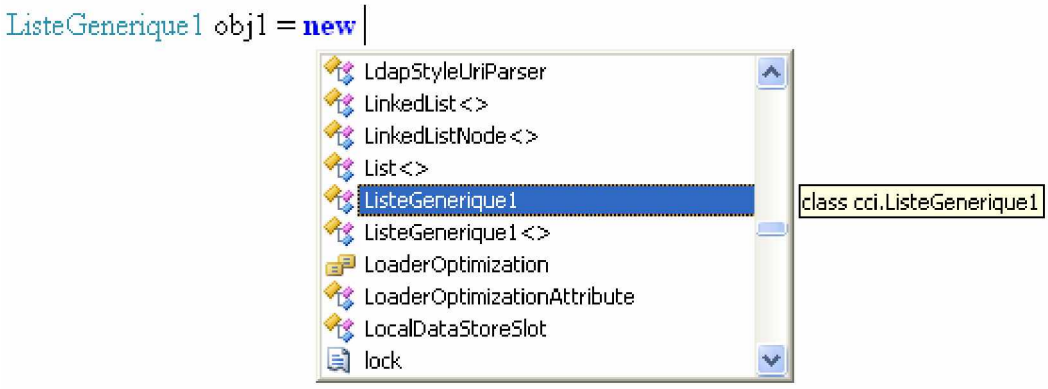
Les types paramétrés font partie intégrante de l'identification d'une classe aussi bien que le nom de la classe elle-même. A l'instar des méthodes qui peuvent être surchargées, il est possible pour une même classe de disposer de plusieurs surcharges : même nom, types paramètres différents.

Voici par exemple **trois classes distinctes** pour le compilateur C#, chacune de ces classes n'a aucun rapport avec l'autre si ce n'est qu'elle porte le même nom et qu'elles correspondent chacune à une surcharge différente de la classe ListeGenerique1 :

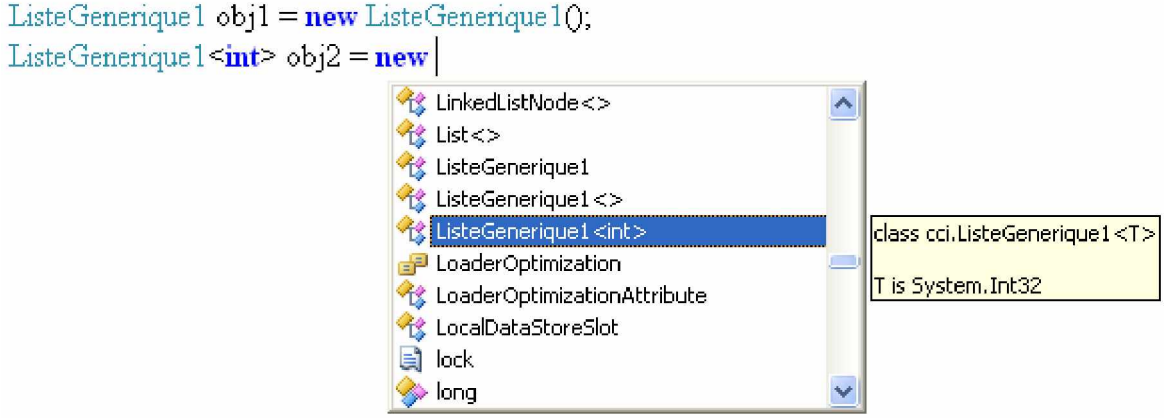
```
public class ListeGenerique1<T>
{
}
public class ListeGenerique1
{
}
public class ListeGenerique1<T, U>
{
}
```

Cet aspect comporte pour le développeur non habitué à la surcharge de classe à un défaut apparent de lisibilité largement compensé par l'audit de code dans Visual C#.

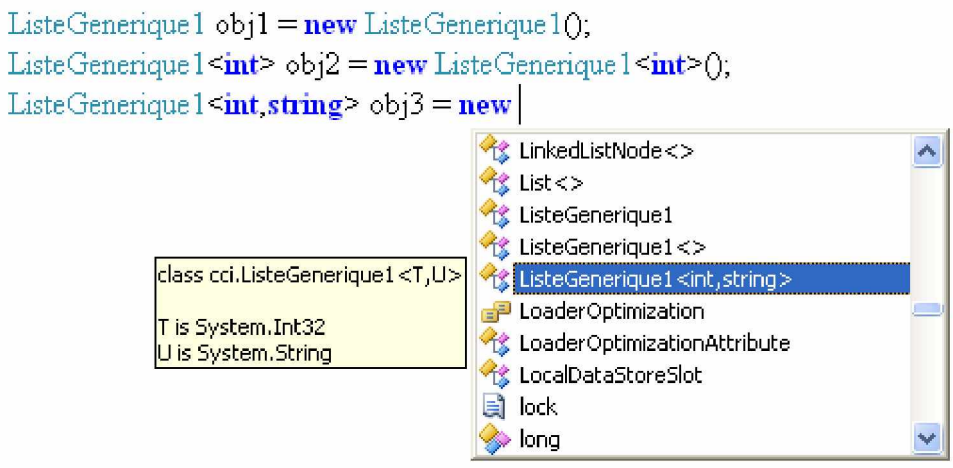
Exemple d'instanciation sur la première surcharge générique :



Exemple d'instanciation sur la seconde surcharge générique :



Exemple d'instanciation sur la troisième surcharge générique :



Les trois instanciations obtenues créent trois objets obj1 , obj2 , obj3 différents et de classe différente :

```
ListeGenerique1 obj1 = new ListeGenerique1();
ListeGenerique1<int> obj2 = new ListeGenerique1<int>();
ListeGenerique1<int, string> obj3 = new ListeGenerique1<int, string>();
```

On peut définir une surcharge générique d'une classe à partir d'une **autre surcharge générique de la même classe** :

```
public class ListeGenerique1<T, U>
{
}
public class ListeGenerique1<T> : ListeGenerique1<T, int>
{
}
```

Il est bien entendu possible de définir des surcharge générique d'une classe à partir d'**autres surcharges génériques d'une autre classe** :

```
public class ListeGenerique1<T, U>    { … }

public class ListeGenerique1<T> : ListeGenerique1<T, int> { … }

public class ListeGenerique2 <U>: ListeGenerique1<string, U> { … }

public class ListeGenerique2 : ListeGenerique1<string, int> { … }

public class ListeGenerique2<T,U> : ListeGenerique1<T> { … }
```

Les types partiels

Plan général:

1. Les types partiels

1. types partiels

Depuis la version 2.0, C# accepte la définition de struct, de classe, d'interface séparées. Ce qui revient à pouvoir définir à plusieurs endroits distincts dans un même fichier, un des trois types précédents ou encore le même type peut voir sa définition répartie sur plusieurs fichiers séparés.

Pour permettre ce découpage en plusieurs morceaux d'un même type, il faut obligatoirement utiliser dans la déclaration et juste avant la caractérisation du type (**struct**, **class**, **interface**), le modificateur **partial**.

Syntaxe:

```
partial class Truc1 { ….. }
partial struct Truc2 { ….. }
partial interface Truc3 { ….. }
```

Les autres modificateurs définissant les qualités du type (**static**, **public**, **internal, sealed, abstract**) doivent se trouver avant le mot clef **partial** :

```
static public partial class Truc1 { ….. }
internal partial struct Truc2 { ….. }
public partial interface Truc3 { ….. }
```

1.1 Déclaration de classe partielle

Nous étudions dans ce paragraphe, les implications de la déclaration d'une classe partielle sur son utilisation dans un programme.

soit un fichier *partie1.cs* contenant la classe ClassPartielle possédant un attribut entier public y initialisé à 200 dans l'espace de nom **cci** :

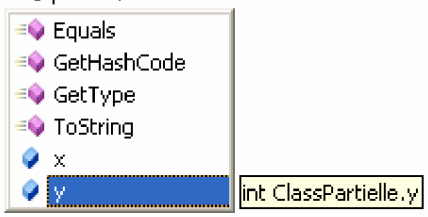
```
namespace cci
{
   partial class ClassPartielle
   {
      public int y = 200;
   }
}
```

On peut définir dans un autre fichier *partie2.cs* la "suite" de la classe ClassPartielle avec un autre attribut **public** x initialisé à 100 dans l'espace de même nom **cci** :

```
namespace cci
{
   partial class ClassPartielle
   {
      public int x = 100;
   }
}
```

Ajoutons dans le code du second fichier *partie2.cs*, une classe Program contenant la méthode **Main**, l'audit de code de C#, nous montre bien qu'un objet de type ClassPartielle, possède bien les deux attributs x et y :

```
namespace cci
{
   partial class ClassPartielle
   {
      public int x = 100;
   }
   class Program
   {
      static void Main(string[] args)
      {
         ClassPartielle Obj = new ClassPartielle();
         Obj.        }
   }
}
```



3 déclarations partielles de la même classe	Implications sur la classe

partial class ClassPartielle {
  public int x = 100;
}
….
sealed partial class ClassPartielle {
  public int y = 200;
}
….
internal partial class ClassPartielle {
  public int z = 300;
}
	La classe est considérée par le compilateur comme possédant la réunion des deux qualificateurs **sealed** et **internal** :
	
	sealed internal partial class ClassPartielle
	{
	  ….
	}

Attention le compilateur C# vérifie la cohérence entre tous les déclarations différentes des qualificateurs d'une même classe partielle. Il décèlera par exemple une erreur dans les

déclarations suivantes :

```
public partial class ClassPartielle {
      public int x = 100;
}
….
internal partial class ClassPartielle {
       public int y = 200;
}
```

Dans l'exemple ci-dessus, nous aurons le message d'erreur du compilateur C# suivant :
Error : Les déclarations partielles de 'ClassPartielle' ont des modificateurs d'accessibilité en conflit.

Conseil : pour maintenir une bonne lisibilité du programme mettez tous les qualificateurs devant chacune des déclarations de classe partielle.

```
sealed internal partial class ClassPartielle {
      public int x = 100;
}
….
sealed internal partial class ClassPartielle {
       public int y =
}
….etc
```

Conseil : ne pas abuser du concept partial, car bien que pratique, la dissémination importante des membres d'une classe dans plusieurs fichiers peut nuire à la lisibilité du programme !

1.2 classe partielle : héritage, implémentation et imbrication

Héritage de classe

Si une classe partielle ClassPartielle hérite dans l'une de ses définitions d'une autre classe ClasseA partielle ou non, cet héritage s'étend implicitement à toutes les autres définitions de ClassPartielle :

```
public class ClasseA {
….
}
public partial class ClassPartielle : ClasseA {
      public int x = 100;
}….
public partial class ClassPartielle {
       public int y = 200;
}….
public partial class ClassPartielle {
      public int z = 300;
}….
```

Conseil : pour maintenir une bonne lisibilité du programme mettez la qualification d'héritage dans chacune des déclarations de la classe partielle.

```
public class ClasseA {
….
}
public partial class ClassPartielle : ClasseA {
      public int x = 100;
}
public partial class ClassPartielle : ClasseA {
       public int y = 200;
}
public partial class ClassPartielle : ClasseA {
      public int z = 300;
}
```

Implémentation d'interface

Pour une classe partielle implémentant une ou plusieurs interfaces le comportement est identique à celui d'une classe partielle héritant d'une autre classe. Vous pouvez écrire :

```
public interface InterfA { … }
public interface InterfB { … }

public partial class ClassPartielle : InterfA {
      public int x = 100;
}….
public partial class ClassPartielle : {
       public int y = 200;
}….
public partial class ClassPartielle : InterfB {
      public int z = 300;
}….
```

Ecriture plus lisible conseillée :

```
public interface InterfA { … }
public interface InterfB { … }

public partial class ClassPartielle : InterfA , InterfB {
      public int x = 100;
}….
public partial class ClassPartielle : InterfA , InterfB {
       public int y = 200;
}….
public partial class ClassPartielle : InterfA , InterfB {
      public int z = 300;
}….
```

Classe imbriquée

Les classes imbriquées de C# peuvent être définies sous forme de classes partielles.

Exemple d'imbrication dans une classe non partielle :

```
class classeMere
{
      partial class classeA
```

```
    {
    }
     …..
     partial class classeA
    {
    }
}
```

Imbrication dans une classe partielle :

```
partial class classeMere
{
     partial class classeA
    {
    }
     …..
     partial class classeA
    {
    }
}
 ……
partial class classeMere
{
     partial class classeA
    {
    }
     …..
     partial class classeA
    {
    }
}
```

1.3 classe partielle : type générique et contraintes

Une classe générique peut être partielle

Soit la classe de liste générique définie plus haut mise sous forme de 3 définitions partielles :

Première définition partielle :

```
public partial class ListeGenerique <T>
{
     List<T> liste = new List<T>();
}
```

Seconde définition partielle :

```
public partial class ListeGenerique <T>
{
     public void ajouter ( T elt )
     {
        this.liste.Add(elt);
     }
}
```

Troisième définition partielle :

```
public partial class ListeGenerique <T>
{
      public T afficher(int rang)
      {
         return this.liste[rang];
      }
}
```

Notons que le ou les types génériques paramétrant la classe partielle doivent obligatoirement être présents, sauf à respecter les conventions de la surcharge générique de classe vue plus haut.

Une classe partielle peut avoir des surcharges génériques partielles

Dans ce cas aussi nous ne saurions que trop conseiller au lecteur d'écrire explicitement dans toutes les définitions partielles d'une même surcharge, les mêmes listes de types paramétrés de la définition.

Notre exemple ci-dessous montre trois surcharges génériques d'une classe ListeGenerique1 : soit ListeGenerique1, ListeGenerique1<T> et ListeGenerique1<T, U>, chaque surcharge générique est déclarée sous forme de deux définitions de classes partielle.

```
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1<T>
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1
{
}
public partial class ListeGenerique1<T, U>
{
}
public partial class ListeGenerique1<T, U>
{
}
```

Types génériques contraints dans les classes partielles

Il semble inutile de jongler avec les exclusions possibles ou non sur la position des différents paramètres de la clause **where** dans plusieurs déclarations d'une même classe partielle, nous proposons de jouer la lisibilité en explicitant la (les) même(s) clause(s) **where** dans chaque déclaration de la classe partielle :

Première déclaration partielle :

```
public class ClasseA { ….}
public interface InterfB { … }
```

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      List<T> liste = new List<T>();
}
```

Seconde déclaration partielle :

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      public void ajouter ( T elt )
      {
         this.liste.Add(elt);
      }
}
```

Troisième déclaration partielle :

```
public partial class ListeGenerique <T>
 where T : ClasseA , InterfB , new( )
{
      public T afficher(int rang)
      {
         return this.liste[rang];
      }
}
```

Les méthodes anonymes

Plan général:

1. Les méthodes anonymes

1. Les méthodes anonymes

Le concept de méthode anonyme dans C# est semblable au concept de classe anonyme en Java très utilisé pour les écouteurs.

L'objectif est aussi dans C# de réduire les lignes de code superflues tout en restant lisible. Les méthodes anonymes en C# sont intimement liées aux délégués.

1.1 Délégué et méthode anonyme

Là où le développeur peut mettre un objet délégué, il peut aussi bien mettre une méthode anonyme. Lorsque dans le programme, il n'est pas nécessaire de connaître le nom de la méthode vers laquelle pointe un délégué, les méthodes anonymes sont alors un bon outil de simplification du code.

Prenons un exemple d'utilisation classique d'un délégué censé permettre fictivement de pointer vers des méthodes de recherche d'un nom dans une liste. Rappelons la démarche générale pour la manipulation de ce concept :

```
 // 1°) la classe de délégué
delegate string DelegListe(int rang);

 class ClassMethodeAnonyme
 {
    // 2°) la méthode vers laquelle va pointer le délégué
    private string searchNom1(int x)
    {
       return "found n° "+Convert.ToString(x);
    }
    public void utilise()
    {
       // 3°) création d'un objet délégué pointant vers la méthode
       DelegListe search1 = new DelegListe(searchNom1);
       Console.WriteLine(search1(1));
    }
 }
```

Reprenons les mêmes éléments mais avec une méthode anonyme :

```
// 1°) la classe de délégué
 delegate string DelegListe(int rang);

 class ClassMethodeAnonyme
 {
    // 2°) la méthode anonyme vers laquelle pointe le délégué
    private DelegListe searchNom2 = delegate ( int x )
    {
       return "found n° "+Convert.ToString(x);
    }
```

```
    public void utilise()
    {
       Console.WriteLine(searchNom2 (2));
    }
 }
```

1.2 Création pas à pas d'une méthode anonyme

Explicitons la démarche ayant conduit à l'écriture du remplacement dans le premier paragraphe précédent des étapes 2° et 3° par la seule étape 2° avec une méthode anonyme dans le second paragraphe.

1°) Une méthode anonyme est déclarée par le mot clef général **delegate** avec son corps de méthode complet :

```
    delegate ( int x )
    {
       return "found n° "+Convert.ToString(x);
    }
```

2°) On utilise une référence de **delegate** , ici searchNom2 de type DelegListe :

```
    private DelegListe searchNom2
```

3°) Une méthode anonyme possède la signature de la classe **delegate** avec laquelle on souhaite l'utiliser (ici DelegListe) directement par l'affectation à une référence de **delegate** appropriée :

```
    private DelegListe searchNom2 = delegate ( int x )
    {
       return "found n° "+Convert.ToString(x);
    }
```

Si le lecteur souhaite exécuter l'exemple complet, voici le code de la méthode Main de la classe principale affichant les résultats fictifs :

```
class Program
{
   static void Main(string[] args)
   {
      ClassMethodeAnonyme obj = new ClassMethodeAnonyme();
      obj.utilise();
      Console.ReadLine();
   }
}
```

1.3 Gestionnaire anonyme d'événement classique sans information

Reprenons un exemple de construction et d'utilisation d'un **événement classique sans information** comme le click sur le bouton dans la fenêtre ci-dessous, et utilisons un gestionnaire d'événements anonyme.

La démarche classique (prise en charge automatiquement dans les environnements Visual Studio et Borland Studio) est la suivante :

- Créer et faire pointer l'événement Click (qui est un **delegate**) vers un nom de gestionnaire déjà défini (ici button1_Click) :

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

- Définir le gestionnaire button1_Click selon la signature du **delegate** :

```
private void button1_Click(object sender, EventArgs e)
{
        textBox1.Text = "click sur bouton-1";
 }
```

L'utilisation d'un **gestionnaire anonyme** réduit ces deux instructions à une seule et au même endroit :

```
this.button1.Click += delegate (object sender, System.EventArgs e)
{
        textBox1.Text = "click sur bouton-1";
};
```

1.4 Gestionnaire anonyme d'événement personnalisé avec information

Soit un exemple d'utilisation de 4 gestionnaires anonymes d'un même événement personnalisé avec information appelé **Enlever**. Pour la démarche détaillée de création d'un tel événement que nous appliquons ici, nous renvoyons le lecteur au chapitre "Evenements" de cet ouvrage.

Nous déclarons une classe d'informations sur un événement personnalisé :

```
public class EnleverEventArgs : EventArgs { … }
```

Nous déclarons la classe delegate de l'événement personnalisé :

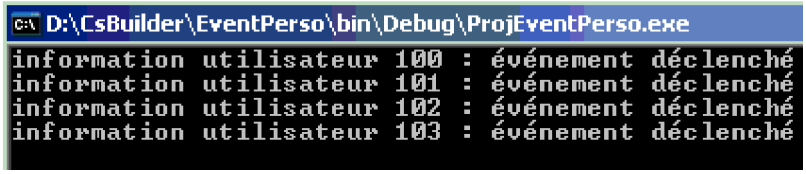
> **public delegate void** DelegueEnleverEventHandler (**object** sender, EnleverEventArgs e);

Nous donnons le reste sans explication supplémentaire autre que les commentaires inclus dans le code source,

```
using System;
using System.Collections;

namespace cci {

   //--> 1°) classe d'informations personnalisées sur l'événement
   public class EnleverEventArgs : EventArgs
   {
      public string info;
      public EnleverEventArgs(string s)
      {
         info = s;
      }
   }
   //--> 2°) déclaration du type délégation normalisé
   public delegate void DelegueEnleverEventHandler(object sender, EnleverEventArgs e);

   public class ClassA
   {
      //--> 3°) déclaration d'une référence event de type délégué :
      public event DelegueEnleverEventHandler Enlever;
      //--> 4.1°) méthode protégée qui déclenche l'événement :
      protected virtual void OnEnlever(object sender, EnleverEventArgs e)
      {
         //....
         if (Enlever != null) Enlever(sender, e);
         //....
      }
      //--> 4.2°) méthode publique qui lance l'événement :
      public void LancerEnlever()
      {
         //....
         EnleverEventArgs evt = new EnleverEventArgs("événement déclenché");
         OnEnlever(this, evt);
         //....
      }
   }

   public class ClasseUse
   {
      //--> 5°) la méthode permettant l'utilisation des gestionnaires anonymes
      static public void methodUse()
      {
         ClassA ObjA = new ClassA();
         ClasseUse ObjUse = new ClasseUse();

         //--> 6°) abonnement et définition des 4 gestionnaires anonymes :
         ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
         {
            //...gestionnaire d'événement Enlever: méthode d'instance.
            System.Console.WriteLine("information utilisateur 100 : " + e.info);
         };
```

```
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
          //...gestionnaire d'événement Enlever: méthode d'instance.
          System.Console.WriteLine("information utilisateur 101 : " + e.info);
        };
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
          //...gestionnaire d'événement Enlever: méthode d'instance.
          System.Console.WriteLine("information utilisateur 102 : " + e.info);
        };
        ObjA.Enlever += delegate(object sender, EnleverEventArgs e)
        {
          //...gestionnaire d'événement Enlever: méthode de classe.
          System.Console.WriteLine("information utilisateur 103 : " + e.info);
        };

        //--> 7°) consommation de l'événement:
        ObjA.LancerEnlever();  //...l'appel à cette méthode permet d'invoquer l'événement Enlever
    }
    static void Main(string[] args)
    {
      ClasseUse.methodUse();
      Console.ReadLine();
    }
  }
}
```

Résultats d'exécution du programme précédent avec gestionnaires anonymes :



1.5 Les variables capturées par une méthode anonyme

Une méthode anonyme accède aux données locales du block englobant dans lequel elle est définie.

- Les variables locales utilisables par une méthode anonyme sont appelées **variables externes**.
- Les variables utilisées par une méthode anonyme sont appelées les **variables externes capturées**.

Exemple :

```
int y =100; // variable externe au bloc anonyme

.... = delegate (int x)
      {
        y = x-3; // variable externe x capturée par le bloc anonyme
      };
....
```

Tous type de variables (valeur ou référence) peut être capturé par un block anonyme.

Le block englobant peut être une méthode :

```
delegate string DelegateListe(int rang);
…….
public void utilise ( )
{
     int entier = 100; //...variable locale type valeur
     object Obj = null; //... variable locale type référence

   DelegateListe searchNum =  delegate(int x)
   {
      entier = 99; //...variable capturée
      Obj = new object ( ); //...variable capturée
      return "found n° " + Convert.ToString(x);
   };
   Console.WriteLine("static : " + searchNum(2));
}
```

Le block englobant peut être une classe :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //...champ de classe type valeur
   int entierInstance = 200;  //...champ d'instance type valeur
   object Obj = null; //... champ d'instance type référence

   public void utiliseMembre ( )
   {
      DelegateListe searchNum =  delegate(int x)
      {
         entierStatic++; //...variable capturée
         entierInstance++;  //...variable capturée
         Obj = new object ( ); //...variable capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(1) );
  }
```

Il est bien sûr possible de combiner les deux genres de variables capturées, soit local, soit membre.

1.6 Les méthode anonymes sont implicitement de classe ou d'instance

Une méthode anonyme peut être implicitement de classe ou bien implicitement d'instance **uniquement**, selon que le délégué qui pointe vers la méthode est lui-même dans une méthode de classe ou une méthode d'instance.

Le fait que le délégué qui pointe vers une méthode anonyme soit explicitement **static**, n'induit pas que la méthode anonyme soit de classe; c'est la méthode englobant la méthode anonyme qui impose son modèle **static** ou d'instance.

a) Exemple avec **méthode englobante d'instance** et **membre délégué d'instance** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   DelegateListe searchNum; //...membre délégué d'instance

   public void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++;//...variable de classe capturée
         entierInstance++; //...variable d'instance capturée
         Obj = new object ( ); //...variable d'instance capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(1) );
  }
}
```

b) Exemple avec **méthode englobante d'instance** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   static DelegateListe searchNum; //...membre délégué de classe

   public void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++;//...variable de classe capturée
         entierInstance++; //...variable d'instance capturée
         Obj = new object ( ); //...variable d'instance capturée
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(2) );
  }
}
```

c) Exemple avec **méthode englobante de classe** et **membre délégué de classe** explicite :

```
delegate string DelegateListe(int rang);
```

```
class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   static DelegateListe searchNum; //...membre délégué de classe

   public static void utiliseMembre ( )
   {
      searchNum =  delegate(int x)
      {
         entierStatic++; //...variable static capturée
         entierInstance++;  ç  erreur de compilation membre d'instance non autorisé
         Obj = new object ( );  ç  erreur de compilation membre d'instance non autorisé
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(3) );
   }
}
```

d) Exemple avec **méthode englobante de classe** et **membre délégué d'instance** explicite :

```
delegate string DelegateListe(int rang);

class ClasseA
{
   static int entierStatic = 100; //... membre de classe type valeur
   int entierInstance = 200;  //... membre d'instance type valeur
   object Obj = null; //... membre d'instance type référence
   DelegateListe searchNum; //...membre délégué d'instance

   public static void utiliseMembre ( )
   {
      searchNum =  delegate(int x)  ç  erreur de compilation le  membre doit être static
      {
         entierStatic++; //...variable static capturée
         entierInstance++;  ç  erreur de compilation membre d'instance non autorisé
         Obj = new object ( );  ç  erreur de compilation membre d'instance non autorisé
         return "found n° " + Convert.ToString(x);
      };
      Console.WriteLine( searchNum(4) );
   }
}
```

Nous remarquons dans les exemples précédents, que le délégué **searchNum** qui pointe vers la méthode anonyme **delegate(int** x) { … } possède bien les mêmes caractéristiques que la méthode englobante **utiliseMembre ( ).** C'est pourquoi dans les exemples (c) et (d) **searchNum** est implicitement **static** comme **utiliseMembre( )** et donc les erreurs signalées par le compilateur dans la méthode anonyme, sont des erreurs classiques commises sur une méthode qui est **static** et qui ne peut accéder qu'à des entités elles-mêmes **static**.

1.7 Comment les méthode anonymes communiquent entre elles

La notion de variable externe capturée par une méthode anonyme permet à plusieurs méthodes anonymes déclarées dans le même block englobant de partager et donc capturer les mêmes variables externes.

Ces variables sont donc comme des variables communes à toutes les méthodes anonymes qui peuvent ainsi communiquer entre elles comme dans l'exemple ci-dessous où deux méthodes anonymes pointées l'une par le délégué **searchNum**, l'autre par le délégué **combienDeSearch** partagent les mêmes variables **numero** (variable locale au bloc englobant) et **nbrDeRecherche** membre d'instance de la classe :

```
class ClasseA
{
   int nbrDeRecherche = 0;  //... membre d'instance type valeur
   DelegateListe searchNum; //...membre délégué d'instance
   DelegateConsulter combienDeSearch; //...membre délégué d'instance

   public void utiliseMembre()
   {
      int numero = 12345;
      searchNum = delegate(int x)
      {
         nbrDeRecherche++;//...variable d'instance capturée
         numero++; //...variable locale capturée
         return "found n° " + Convert.ToString(x);
      };
      combienDeSearch = delegate()
      {
         return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
                  + " , numéro : " + Convert.ToString(numero);
      };
      Console.WriteLine( searchNum(50) );
      Console.WriteLine( combienDeSearch( ) );
   }
}
```

Résultats obtenu lors de l'exécution sur la console :
found n° 50
nombre de recherches effectuées : 1 , numéro : 12346

Attention : une méthode anonyme ne s'exécute que lorsque le délégué qui pointe vers elle est lui-même invoqué, donc les actions effectuées sur des variables capturées ne sont effectives que lors de l'invocation du délégué.

Si nous reprenons la classe précédente et que nous reportions l'initialisation "**numero = 12345;**"de la variable locale **numero** dans la première méthode anonyme, nous aurons un message d'erreur du compilateur :

```
class ClasseA
{
   int nbrDeRecherche = 0;  //... membre d'instance type valeur
   DelegateListe searchNum; //...membre délégué d'instance
   DelegateConsulter combienDeSearch; //...membre délégué d'instance
```

```
    public void utiliseMembre()
    {
       int numero; //...variable locale non initialisée
       searchNum = delegate(int x)
       {
          nbrDeRecherche++;//...variable d'instance capturée
          numero= 12345; //...variable locale capturée et initialisée
          return "found n° " + Convert.ToString(x);
       };
       combienDeSearch = delegate()
       {
          return "nombre de recherches effectuées : " + Convert.ToString(nbrDeRecherche)
                 + " , numéro : "
                 + Convert.ToString(numero);  ç  erreur de compilation variable numero non initialisée !
       };
       Console.WriteLine( searchNum(50) );
       Console.WriteLine( combienDeSearch( ) );
    }
}
```