

Les génériques

Les génériques sont des éléments de programme capables de s'adapter de manière à fournir les mêmes fonctionnalités pour différents types de données.

Ils ont été introduits avec l'arrivée du framework .NET 2.0 dans l'objectif de fournir des services adaptés à plusieurs types de données tout en gardant un typage fort.

Les frameworks .NET 1.0 et .NET 1.1 fournissaient la classe `ArrayList` pour la gestion de listes dynamiques. Ce type, quoique très pratique, avait l'inconvénient de ne contenir que des objets de type `System.Object`, ce qui induisait un nombre important de transtypes dont il était souhaitable de se passer. Les génériques sont la réponse apportée par Microsoft à ce problème : la classe `List<T>` remplace avantageusement (dans la plupart des cas) la classe `ArrayList` en permettant de définir le type des objets qu'elle contient.

Ceci simplifie le code tout en le rendant plus sûr puisque le développeur élimine une grosse partie des problèmes de transtypage potentiels.

Les éléments génériques sont reconnaissables aux caractères `<` et `>` présents dans leurs noms. Ces symboles encadrent les noms de types associés à une instance d'un élément générique.

Les génériques seront ici étudiés au travers du développement d'un type permettant de gérer une file d'éléments. Les files sont des collections suivant le principe FIFO (*First In, First Out*) : il n'est possible d'accéder qu'au premier élément de la collection, et lorsque l'on ajoute un élément, celui-ci est automatiquement placé en dernière position de la collection. Un type similaire existe dans le framework .NET : `System.Collections.Generic.Queue<T>`.

1. Classes

Les classes génériques sont conçues pour manipuler d'une manière unifiée plusieurs types de données. Ces types sont désignés par un ou plusieurs alias dans le nom de la classe, et les types concrets sont assignés à ces alias à l'instanciation d'un objet générique.

a. Définition d'une classe générique

Pour créer une classe générique, il faut définir une classe, lui adjoindre les caractères `<` et `>` et placer entre ceux-ci un ou plusieurs alias de types séparés par des virgules. Ces alias permettent de manipuler des variables fortement typées sans en connaître préalablement le type concret.

```
public class ListeFIFO<TElement>
{
}
```



Par convention, les alias de type sont préfixés par la lettre T (pour Type).

La déclaration d'une classe générique ayant la capacité de gérer deux types de données simultanément pourrait être écrite de la manière suivante :

```
public class ListeDoubleFIFO<TElement1, TElement2>
{
}
```

b. Utilisation d'une classe générique

Pour instancier un type générique, Il faut utiliser l'opérateur `new` suivi du nom du type générique. Les alias de type doivent être remplacés par des noms de types concrets.

```
ListeFIFO<string> liste = new ListeFIFO<string>();
```

Cette instantiation indique que chaque élément marqué par l'alias de type `TElement` dans la classe `ListeFIFO` sera traité comme étant de type `string`. Ce comportement n'est valable que pour l'instance définie ici. Il est parfaitement possible de définir une autre instance manipulant des entiers ou des fichiers (`System.IO.File`).



Il est aussi possible de définir des structures génériques. Leur déclaration est similaire à celle des classes génériques. La différence est dans l'utilisation du mot-clé `struct` en lieu et place du mot-clé `class`.

2. Interfaces

Une interface générique permet de définir un contrat d'utilisation indépendant du type de données manipulé par les classes qui l'implémentent.

a. Définition d'une interface générique

La définition d'une interface générique est semblable à la définition d'une interface normale. La différence réside dans le fait qu'il est nécessaire d'associer un ou plusieurs types paramètres après le nom de l'interface, comme pour la déclaration d'une classe générique.

```
public interface IListeFIFO<TElement>
{
}
```



La convention d'écriture de C# indique que le nom des interfaces, qu'elles soient génériques ou non, commence par un `I` (i majuscule).

La définition des membres de l'interface peut faire référence à chacun des types paramètres de l'interface, sans obligation de le faire. En ajoutant deux signatures de méthodes à l'interface pour ajouter et enlever un élément, la définition du type `IListeFIFO<TElement>` ressemble à ceci :

```
public interface IListeFIFO<TElement>
{
    void AjouterElement(TElement elementAAjouter);

    TElement EnleverElement();
}
```

Ces signatures utilisent toutes deux le type paramètre `TElement`. Une classe qui implémente cette interface en précisant que le type paramètre est `string` (`IListeFIFO<string>`) doit implémenter ces méthodes de la

manière suivante :

```
public void AjouterElement(string elementAAjouter)
{
    //implémentation concrète de la méthode
}

public string EnleverElement()
{
    //implémentation concrète de la méthode
}
```

b. Utilisation d'une interface générique

Les interfaces génériques sont utilisées de la même manière que les interfaces normales, mais il faut évidemment spécifier quels sont ses types paramètres.

```
class ListeEntiers : IListeFIFO<int> {
    public void AjouterElement(int elementAAjouter)
    {
        //implémentation concrète de la méthode
    }

    public int EnleverElement()
    {
        //implémentation concrète de la méthode
    }
}
```

Dans le cas de la classe `ListeFIFO<TElement>`, il est pertinent que le type paramètre de l'interface soit le même que celui utilisé lors de l'instanciation d'un objet. Il est tout à fait possible d'écrire la déclaration de classe suivante pour arriver à ce résultat.

```
public class ListeFIFO<TElement> : IListeFIFO<TElement>
```

L'implémentation complète (et non optimisée) de la classe à ce stade peut être la suivante :

```
public class ListeFIFO<TElement> : IListeFIFO<TElement>
{
    private TElement[] tableauInterne;

    public ListeFIFO()
    {
        tableauInterne = new TElement[0];
    }

    public void AjouterElement(TElement elementAAjouter)
    {
```

```

        TElement[] tableauTemporaire = new
TElement[tableauInterne.Length + 1];

        tableauInterne.CopyTo(tableauTemporaire, 0);
        tableauTemporaire[tableauInterne.Length] = elementAAjouter;

        tableauInterne = tableauTemporaire;
    }

    public TElement EnleverElement()
    {
        TElement resultat = tableauInterne[0];

        TElement[] tableauTemporaire = new
TElement[tableauInterne.Length - 1];
        for (int i = 1; i < tableauInterne.Length; i++)
        {
            tableauTemporaire[i - 1] = tableauInterne[i];
        }

        tableauInterne = tableauTemporaire;

        return resultat;
    }
}

```

3. Contraintes

Les éléments génériques ont la capacité de s'adapter à la manipulation de plusieurs types de données. Il peut toutefois être pertinent de restreindre le jeu de types paramètres utilisables pour satisfaire certains besoins logiques ou techniques. Le compilateur est capable de vérifier que les types paramètres d'éléments génériques répondent bien aux contraintes fixées. Lorsque l'une des contraintes n'est pas respectée, une erreur de compilation est générée.

Cinq types de contraintes peuvent être fixés sur les types paramètres et peuvent être combinés.

where TParametre : class

Cette contrainte spécifie que le type paramètre doit être un type référence : classe, tableau ou délégué.

```

public class ListeFIFO<TElement> where TElement : class
{
}

```



where TParametre : struct

Cette contrainte impose l'utilisation d'un type valeur (donc non nullable) comme type paramètre.

```
public class ListeFIFO<TElement> where TElement : struct
{
}
```



where TParametre : new()

Lorsque cette contrainte est appliquée, le type paramètre concerné doit avoir un constructeur sans paramètre public. Elle est utilisée lorsque la classe générique doit pouvoir instancier des objets du type paramètre.

```
public class ListeFIFO<TElement> where TElement : new()
{
}
```



where TParametre : <type>

Cette contrainte indique que le type paramètre concerné doit être le type spécifié ou en dériver.

```
public class ListeFIFO<TElement> where TElement : Attribute
{
}
```



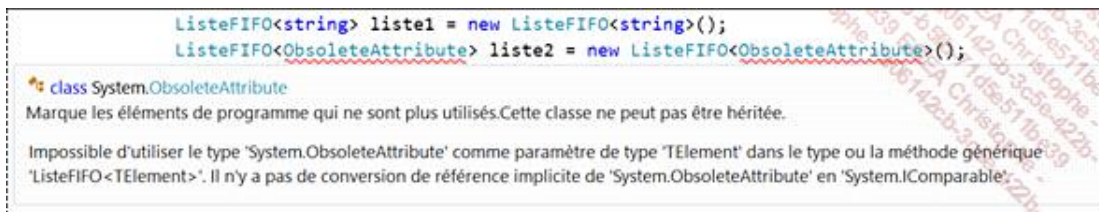
Lorsque le type générique possède plusieurs types paramètres, il est parfaitement envisageable de définir une contrainte sur l'un d'eux indiquant que son type doit être ou doit dériver d'un autre type paramètre.

```
public class CoupleDeValeurs<TValeur1, TValeur2> where TValeur2 :  
    TValeur1  
{  
}  
}
```

where TParametre : <interface>

Cette dernière contrainte force l'utilisation d'un type paramètre implémentant une interface particulière.

```
public class ListeFIFO<TElement> where TElement : IComparable  
{  
}  
}
```



Combinaison de contraintes

Il est possible de combiner les contraintes de manière à obtenir un contrôle plus fin sur les types paramètres.

```
public class ListeFIFO<TElement> : IListeFIFO<TElement> where  
    TElement : class, IComparable, new()  
{  
}  
}
```

Certaines contraintes sont incompatibles entre elles ou utilisables une seule fois par type paramètre. Lorsqu'un cas problématique est décelé, une erreur de compilation est levée.



4. Méthodes

Les méthodes génériques sont des méthodes possédant un ou plusieurs types paramètres. Elles peuvent ainsi effectuer le même traitement sur plusieurs types de données.

Ces méthodes peuvent être définies dans des types génériques ou non, ce qui peut s'avérer pratique pour l'écriture

de méthodes génériques utilitaires placées dans des classes statiques non génériques.

a. Définition d'une méthode générique

Comme pour la définition d'une classe ou d'une interface, les méthodes génériques sont très similaires à leur contrepartie non générique. La différence réside dans l'ajout de types paramètres à leur signature.

La classe `ListeFIFO` peut ainsi fournir une méthode générique `EnleverElement<TResultat>()` renvoyant un élément transtypé du type paramètre de la classe vers le type paramètre de la méthode.


```
public TResultat EnleverElement<TResultat>()
{
    //L'alias de type TElement est défini au niveau de la classe
    //On utilise la méthode non générique EnleverElement pour récupérer
    //un élément de type TElement
    TElement element = EnleverElement();

    //On effectue le transtypage en utilisant l'alias de type TResultat
    //comme type "destination"
    TResultat resultat = (TResultat)element;

    return resultat;
}
```

Malgré son apparente simplicité, le code de cette méthode génère une erreur de compilation au niveau du transtypage :

```
TResultat resultat = (TResultat)element;
```



Cette erreur signifie que le compilateur ne connaît pas de point commun entre les deux types permettant une conversion. Pour résoudre ce problème, il convient d'ajouter une contrainte sur la méthode générique afin de spécifier que `TResultat` est un type dérivé de `TElement`, ce qui permet au compilateur de valider la justesse du transtypage.

```
public TResultat EnleverElement<TResultat>() where TResultat :
TElement
{
    TElement element = EnleverElement();
    TResultat resultat = (TResultat)element;

    return resultat;
}
```

b. Utilisation d'une méthode générique

Les méthodes génériques sont utilisées de la même manière que les méthodes normales. Comme tous les éléments génériques, il suffit de leur adjoindre un ou plusieurs types paramètres afin de les utiliser.

```
//Instanciation d'une liste d'objets.
var liste = new ListeFIFO<object>();
//Ajout d'un objet de type string
liste.AjouterElement("chaîne 1");

//Récupération de l'objet sous la forme d'un string à l'aide
//de la méthode générique EnleverElement<TResultat>
string chaine = liste.EnleverElement<string>();

Console.WriteLine("Mon objet est la chaîne '{0}'", chaine);
```

5. Événements et délégués

De la même manière que des alertes sont levées lorsque les files d'attente deviennent trop longues aux caisses d'un supermarché, il peut être intéressant de générer un événement lorsque le nombre d'éléments devient trop important dans notre liste. Ici, l'événement sera déclenché lorsque la liste contiendra plus de dix éléments.

Tout d'abord, il faut définir un délégué qui prendra comme paramètre l'élément ajouté ayant déclenché l'alerte.

```
public delegate void GestionnaireAlerteNombre<TElementAjoute>(
    TElementAjoute dernierElementAjoute)
```

L'événement est déclaré de la manière suivante :

```
public event GestionnaireAlerteNombre<TElement>
    AlerteNombreElements;
```

Le type `TElement` utilisé dans la déclaration de cet événement est celui utilisé comme type paramètre pour la classe. Le type utilisé dans le gestionnaire d'événements sera ainsi toujours adapté à l'objet `ListeFIFO` instancié.

Une fois l'événement créé, il faut le déclencher quand le nombre d'éléments devient trop important. Pour cela, on définit une méthode dont le but est de déclencher l'événement si un gestionnaire lui est associé, et on modifie le code de la méthode `AjouterElement` pour l'appeler si besoin.

```
protected virtual void DeclencheAlerteNombre(TElement
dernierElementAjoute)
{
    GestionnaireAlerteNombre<TElement> handler =
    AlerteNombreElements;

    //Si un ou plusieurs gestionnaires sont définis,
    //handler ne sera pas null
    if (handler != null)
    {
        handler(dernierElementAjoute);
    }
}
```



```
}  
}
```

Enfin, nous pouvons associer un gestionnaire à une instance de la liste et vérifier que l'événement est lancé.

```
static void Main(string[] args)  
{  
    ListeFIFO<string> liste = new ListeFIFO<string>();  
    //Ajout du gestionnaire d'événements  
    liste.AlerteNombreElements += liste_AlerteNombreElements;  
  
    for (int i = 0; i < 10; i++)  
    {  
        liste.AjouterElement("chaîne " + i);  
    }  
    Console.WriteLine("10 éléments sont dans la liste.");  
    Console.WriteLine("Ajout d'un élément perturbateur...");  
    liste.AjouterElement("élément perturbateur");  
  
    Console.ReadLine();  
}  
  
static void liste_AlerteNombreElements(string dernierElementAjoute)  
{  
    Console.WriteLine("Il y a plus de 10 éléments dans la liste !  
Le dernier est '{0}'", dernierElementAjoute);  
}
```

Ce code affiche le résultat suivant :

```
10 éléments sont dans la liste.  
Ajout d'un élément perturbateur...  
Il y a plus de 10 éléments dans la liste ! Le dernier est  
'élément perturbateur'
```