

Framework .NET : Notions fondamentales

Sommaire

Framework .NET : Notions fondamentales.....	1
1 Introduction.....	2
2 Framework .NET et outils de développement.....	3
2.1 L'intérieur du Framework .NET.	3
2.2 Les outils de développement	5
2.3 Créer son projet.....	5
3 La base d'un logiciel : Les variables	7
3.1 Les variables de type valeur	8
3.1.1 Les types de valeur Built-In	9
3.1.2 Les types de valeur personnalisés	10
3.1.3 Les énumérations	12
3.2 Les variables de références	13
3.2.1 Quelques types références Built-In	15
3.3 Les conversions de types.....	19
3.3.1 Conversions de types.....	19
3.3.2 Le Boxing et l'Unboxing	21
3.3.3 Implémentation des conversions dans les types personnalisés	22
4 Les particularités de la programmation orientée objet	27
4.1 Héritage	27
4.2 Les interfaces.....	29
4.3 Les classes partielles.....	30
4.4 Les classes génériques.....	31
4.5 Les évènements.....	33
4.6 Les attributs.....	35
4.7 Le TypeForwarding	35
5 Conclusion	37

1 Introduction

Le .NET Framework permet le développement d'applications fonctionnelles sur machine Windows équipée de ce Framework. Malgré la simplicité de développement apparente, il n'en est pas moins complexe à connaître et à maîtriser tant les outils proposés sont nombreux.



Au cours de ces 15 modules, vous apprendrez à vous servir des outils de base du Framework en version 2.0 afin de développer des applications facilement, sans pour autant mettre de côté l'optimisation ou la sécurité de celle-ci.

Dans ce chapitre, nous commencerons par établir les bases du Framework en indiquant les outils disponibles ainsi que les notions fondamentales du Framework .NET.

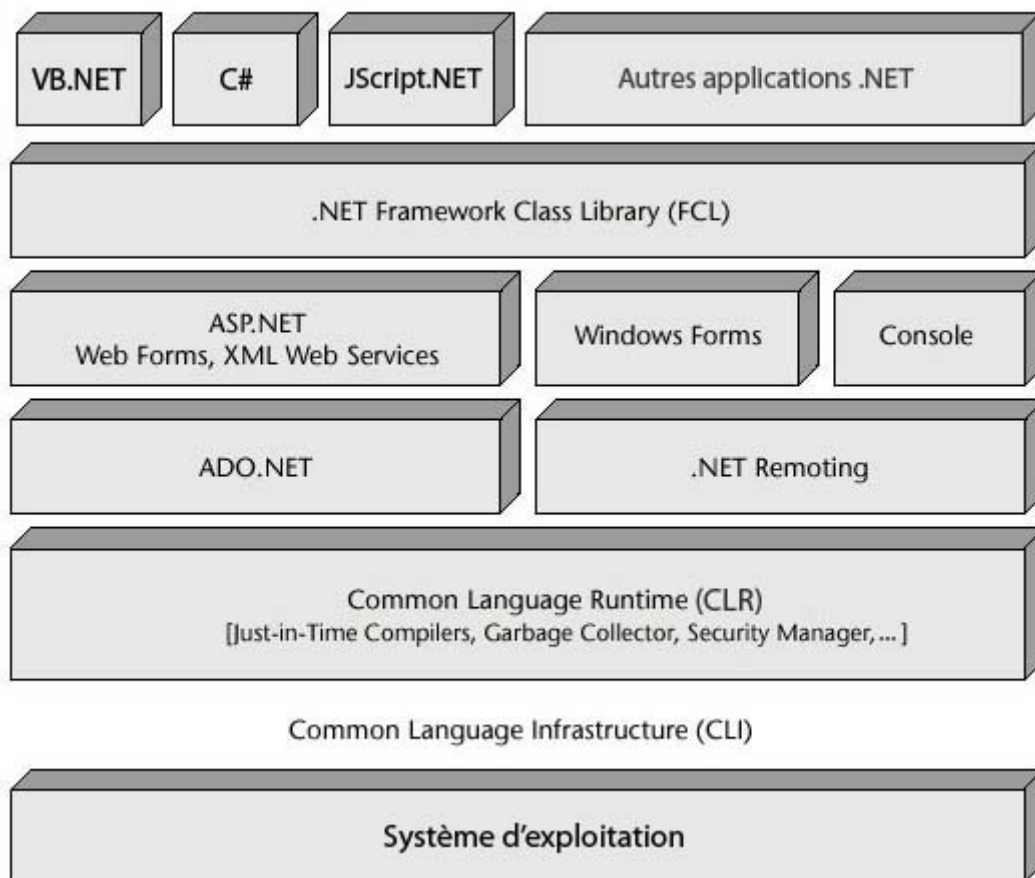
2 Framework .NET et outils de développement

2.1 L'intérieur du Framework .NET.

Afin de suivre ces modules, vous devrez être équipé d'une machine possédant Windows XP ou Vista avec le Framework .NET 2.0 ou supérieur installé.

Note : Il est également possible de développer des logiciels grâce aux technologies .NET sous GNU/Linux et Mac OS X en utilisant Mono. Pour plus d'informations, visitez le site du [projet Mono](#).

Vous pouvez télécharger gratuitement le .NET Framework 3.5 sur [le centre de téléchargement Microsoft](#).



Le Framework .NET est composé de plusieurs parties :

- Une partie « Machine Virtuelle » appelée Common Language Runtime qui comprend le compilateur Just-in-Time (chargé de compiler le code MSIL à l'exécution de l'application), le GarbageCollector (ou « Ramasse miettes », qui a pour but de libérer la mémoire des objets qui ne sont plus référencés dans le programme au cours de son exécution) etc. et qui se charge d'exécuter votre application dans un environnement dit « managé » et sécurisé, de la même manière sur toutes les machines équipées du Framework.
- Une partie bibliothèque qui regroupe une foule de fonctionnalités telles que l'ASP.NET (pour les sites web), l'ADO.NET (pour l'échange de données), la Framework Class Library (pour tous les traitements de bas niveau. C'est ce que nous étudierons dans ces chapitres) etc.
- L'ensemble des langages portés sur le Framework .NET.

Dans le temps, le Framework .NET a subi plusieurs améliorations :

	Fx 1.0	Fx 1.1	Fx 2.0	Fx 3.0	Fx 3.5
Engine Core	CLR 1.0	CLR 1.1	CLR 2.0		
	VB .NET, C#...	VB .NET, C#, C++, J#...	VB .NET, C#, C++, J#... Generics...	LINQ, C#, ...	
Functional Libraries	ADO .NET 1.0 ASP .NET 1.0 XML ASMX ...	ADO .NET 1.1 ASP .NET 1.1 XML ASMX ... WSE 2.0	ADO .NET 2.0 ASP .NET 2.0 XML ASMX ... WSE 3.0	Fx 2.0 + WCF WPF WF WCS	Fx 3.0 + AJAX .ASP.NET
Platform Technology	Windows 98 to Windows XP	+ Windows Server 2003	+ SQL 2005	Win XP SP2, WinSrv 2k3 SP1, Vista	Win XP SP2, WinSrv 2k3 SP1, Vista .SPL, WinSrv 2008, SQL 2008
Developer tools (Visual Studio)	.NET 2002	.NET 2003	2005 And VSTS	2005 + Add-ins	Orcas

L'évolution majeure fut réalisée lors du passage du .NET Framework 1.1 au .NET Framework 2.0. Tous les outils implémentés jusque là ont été repris et améliorés (y compris la CLR) afin d'accroître les performances et la stabilité des applications créées ainsi que leur sécurité.

Le tableau ci-dessus nous montre également qu'à partir de la version 2.0 du Framework, la base de ce dernier reste la même ; seuls des outils annexes y sont ajoutés (comme WCF, WPF, AJAX etc.)

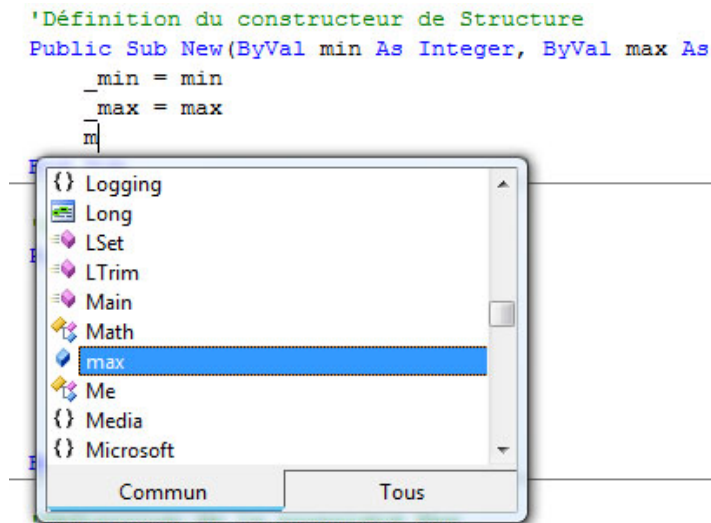
C'est pourquoi tout ce qui sera abordé dans ces modules restera valable, que vous développiez sur le Framework 2.0 ou sur le Framework 3.5.

2.2 Les outils de développement

Pour développer une application facilement, nous utiliserons la suite logicielle Microsoft Visual Studio 2008 qui permet de créer des projets aussi bien en C# qu'en VB.NET.

Cet IDE (Integrated Development Environment) propose, en plus de la coloration syntaxique, une fonctionnalité non négligeable appelée « IntelliSense ». Cette fonctionnalité assez puissante va ajouter à l'éditeur de code une fonction d'auto-complétion qui sélectionnera parmi la liste des fonctionnalités disponibles celle qui convient le mieux et ce, juste en tapant la première lettre du nom de l'objet que vous souhaitez ajouter.

Visual Studio 2008 étant une solution payante et couteuse, Microsoft propose également en téléchargement des versions gratuites de ses IDE, à savoir Visual C# 2008 Express Edition, si vous voulez faire du C#, et Visual Basic 2008 Express Edition, si vous souhaitez faire du VB.NET.



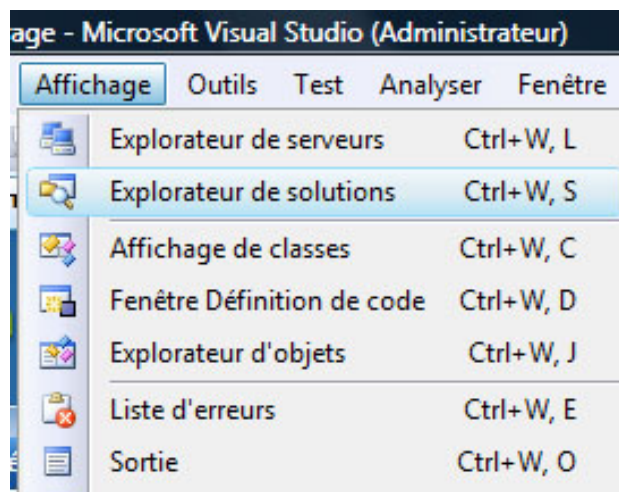
Vous pouvez télécharger toutes ces versions gratuites sur le site Microsoft.com.

2.3 Créer son projet

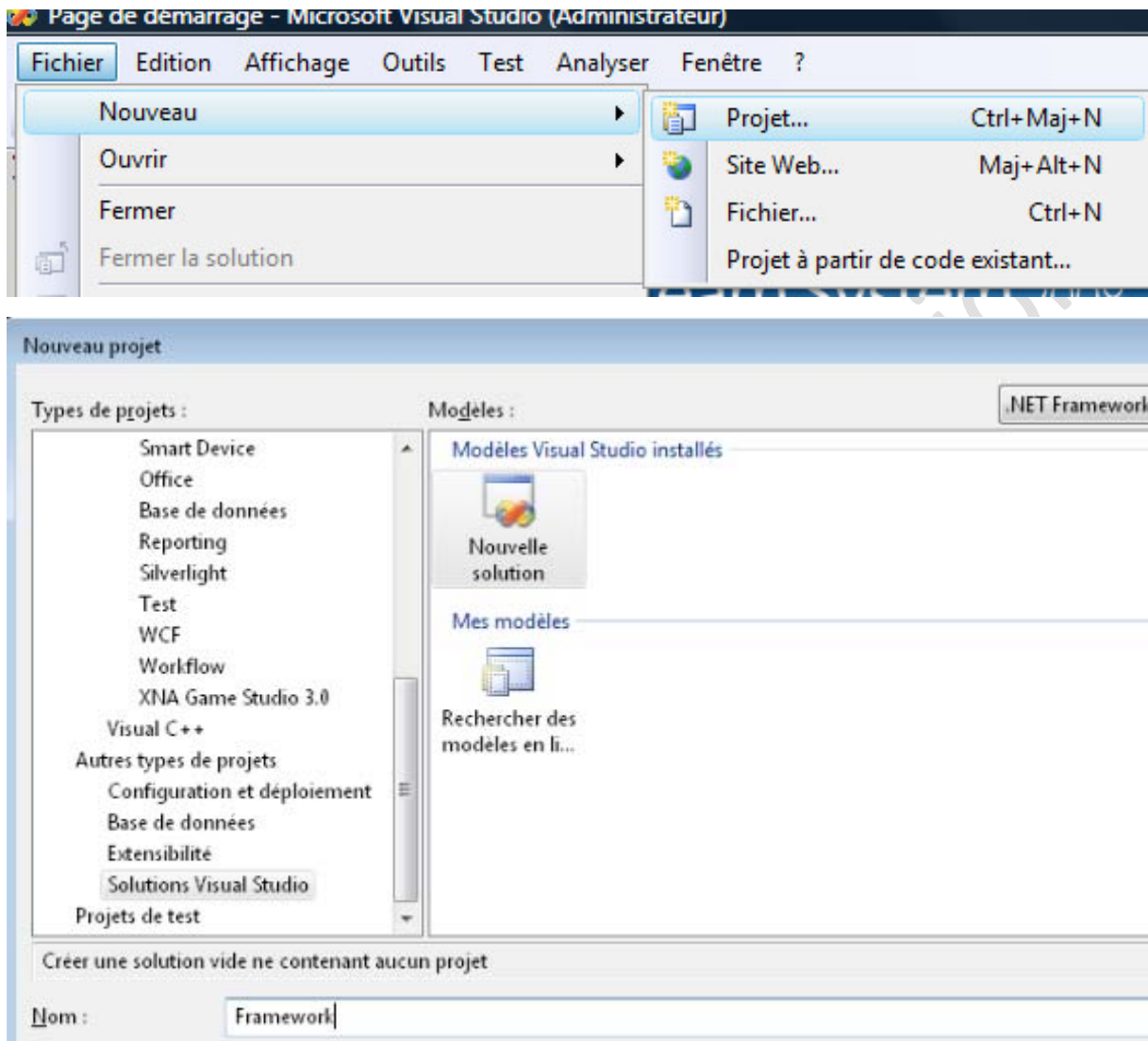
La suite de développement Visual Studio fonctionne sur un principe de « solution » contenant des « projets », lesquels contiennent à leur tour le code de votre application. Par exemple dans le cas d'un Jeu Vidéo, nous aurons notre solution Heroes Racing, contenant un projet Jeu et un projet Moteur Physique.

Avec cette manière de faire, vous pourrez à chaque chapitre créer un projet, et revenir à tout moment sur l'exemple d'un chapitre précédent en quelques clics.

Au lancement de Visual Studio, vous veillerez à afficher l'explorateur de solution en cliquant sur « Affichage → Explorateur de solutions ». Celui-ci apparaîtra à droite de la fenêtre de développement.



Vient ensuite la création du projet. Quelque soit l'IDE que vous avez choisi, la démarche à suivre est sensiblement la même. Cliquez sur « Fichier → Nouveau Projet » et sélectionnez Autre type de projets dans la barre de gauche, puis Solutions Visual Studio, donnez lui un nom (ici Framework).

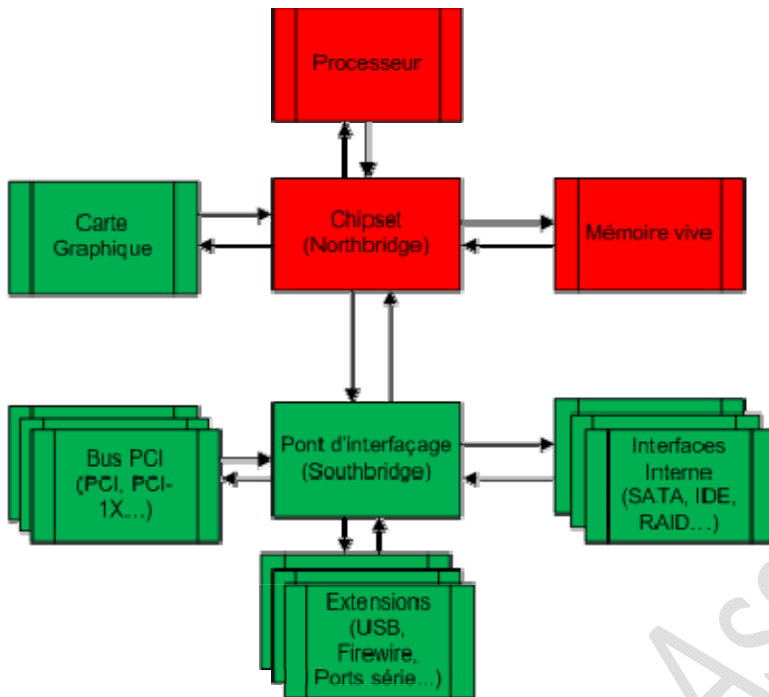


Vous validerez en cliquant sur « Ok ». Votre projet apparaîtra à droite dans l'explorateur de solution et au centre de l'IDE, vous aurez une page de code ajoutée par défaut lors de la création d'un projet.

Lorsque vous souhaitez ajouter un projet à la solution actuellement ouverte, vous devrez cliquer sur « Fichier→Ajouter→Nouveau projet »

3 La base d'un logiciel : Les variables

Afin de comprendre la différence entre les différents types de variable, il faut d'abord garder un aperçu de ce qui se trouve dans nos machines :



Comme illustré ci-contre, un ordinateur possède plusieurs composants. Les composants qui nous intéressent dans cette partie sont en rouge.

Pour s'exécuter, notre application va donc demander au processeur d'effectuer des opérations sur des valeurs qui seront stockées dans la mémoire vive, le tout transitant par une puce de la carte mère appelée « Northbridge » (même si la tendance est d'intégrer aux processeurs actuels un contrôleur de mémoire)

Il faut comprendre que lorsque le .NET Framework exécute une application, il va segmenter la mémoire vive en deux parties distinctes :



3.1 Les variables de type valeur

Lors de l'exécution du programme, celui-ci a un accès total à la pile mémoire. C'est dans cette zone de la mémoire que le programme stockera toutes les variables de type valeur.

Il existe trois sortes de type valeur :

- Les types dit « Built-In », inclus avec le .NET Framework
- Les types personnalisés par le développeur
- Les énumérations

Tous ces types valeur dérivent (directement ou implicitement) de la classe `System.ValueType` qui dérive elle-même de la classe `System.Object` donnant ainsi accès aux méthodes de ces deux classes (notamment la très intéressante méthode `ToString` qui vous permettra entre autre de transformer une valeur numérique en chaîne de caractères).

En pratique, les variables de type valeurs doivent obligatoirement être assignées (contenir quelque chose en mémoire) avant d'être utilisées. Il y a cependant une alternative permettant de mettre l'état de la variable à "null", c'est-à-dire qui ne contient aucune valeur définie. Pour ce faire, il suffit de déclarer nos variables "Nullable":

```
'VB
Dim variable As Nullable(Of Boolean) = Nothing

//C#
Nullable<bool> variable = null;
//Ou
bool? variable = null;
```

Dans tous les cas, la variable déclarée ici est un booléen qui ne contient ni `true`, ni `false` mais `null`.

Il faut également savoir que Microsoft recommande à ces types valeur de ne représenter qu'une seule valeur, d'avoir une taille en mémoire inférieure à 16 octets, de ne pas être modifiée après instantiation et de ne pas être casté en une variable de type référence.

3.1.1 Les types de valeur Built-In

Ce sont tous les types de bases tels que int, float, double, short, char Chacun d'entre eux utilise une certaine taille en mémoire et sont en réalité des alias vers leurs classes respectives dans l'espace de nom System. Voici un récapitulatif des plus utilisés :

Classe	Alias	Octets	Valeur codées
System.SByte	sbyte	1	-128 à 127
System.Byte	byte	1	0 à 255
System.Int16	short	2	-32768 à 32767
System.Int32	int	4	-2147483648 à 2147483647
System.UInt32	uint	4	0 à 4294967295
System.Int64	long	8	-9223372036854775808 à 9223372036854775807
System.Single	float	4	$-3,4^E+38$ à $3,4^E+38$
System.Double	double	8	$-1,79^E+308$ à $1,79^E+308$
System.Decimal	decimal	16	$-7,9^E+29$ à $7,9^E+29$
System.Char	char	2	Caractère Unicode (UTF-16)
System.Boolean	bool	4	Valeur <code>true</code> ou <code>false</code>
System.IntPtr		Dépend de l'OS	Pointeur en mémoire
System.DateTime	date	8	Codage d'une date

Note : Préférez l'utilisation des types int et double (signés ou non signés) lors de traitements nécessitant de nombreuses opérations (compteurs, calcul à virgule flottante) ; ceux-ci sont optimisés par la CLR.

Il existe plus de 300 types valeurs disponibles dans le .NET Framework 2.0.

Le fonctionnement de ces types est strictement identique à celui des variables en C :

- Quand vous effectuez des opérations sur ces variables, leur contenu est directement modifié en mémoire.
- Quand vous passez ces valeurs dans une méthode (à moins de spécifier le mot clef ref ou out), vous travaillerez sur une copie de la valeur et non pas sur la valeur directement.
- Le fait d'assigner une variable de type valeur avec le contenu d'une autre variable de type valeur va dupliquer le contenu en mémoire.

3.1.2 Les types de valeur personnalisés

C'est le nom savant donné aux structures. Ces portions de codes permettent de rassembler sous un même nom personnalisable, plusieurs autres types (valeurs ou références). Par exemple, la structure « Point » sera composée de deux entiers X et Y indiquant la position en abscisse et en ordonnée du point :

```
'VB
Structure Point
    Public _x, _y As Integer
End Structure

//C#
struct Point
{
    public int X;
    public int Y;
}
```

Comme en C, la taille en mémoire des structures est égale à la somme des tailles occupées par les valeurs utilisées dans la structure. Ici, notre structure Point occupera donc une taille en mémoire de $4+4 = 8$ octets.

Afin d'illustrer les concepts abordés dans ces deux parties sur les types valeurs, voici une structure Personnage qui contient deux attributs Age et Nom:

```
'VB
Structure Personnage
    Public Age As Integer
    Public Nom As String
    Public Sub New(ByVal nom As String, ByVal age As Integer)
        Me.Age = age
        Me.Nom = nom
    End Sub
    Public Overrides Function ToString() As String
        Return Nom + " = " + Age.ToString()
    End Function
    Public Shared Operator +(ByVal p1 As Personnage, ByVal p2 As
Personnage) As Personnage
        p1.Nom = p1.Nom + " et " + p2.Nom
        p1.Age = p1.Age + p2.Age
        Return p1
    End Operator
End Structure
```

```
//C#
struct Personnage
{
    public int Age;
    public string Nom;
    public Personnage(string nom, int age)
    {
        this.Nom = nom;
        this.Age = age;
    }
    public override string ToString()
    {
        return Nom.ToString() + " = " + Age.ToString() + " ans";
    }
    public static Personnage operator +(Personnage p1, Personnage p2)
    {
        p1.Nom = p1.Nom + " et " + p2.Nom;
        p1.Age += p2.Age;
        return p1;
    }
}
```

Maintenant que nous possédons cette structure, afin d'observer la duplication des données en mémoire, nous allons déclarer deux variables du type Personnage. Ensuite, nous placerons l'instance de la première variable dans la seconde et nous afficherons le contenu des deux. Puis, on modifiera le contenu de la première avant de visualiser à nouveau le contenu des deux:

```
'VB
Sub Main()
    Dim Paul As Personnage = New Personnage("Pas Paul", 20)
    Dim Lucie As Personnage
    Lucie = Paul
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom)
    Paul.Nom = "Paul"
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom)
    Console.Read()
End Sub

//C#
public string main()
{
    Personnage Paul = new Personnage("Pas Paul", 20);
    Personnage Lucie;
    Lucie = Paul;
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom);
    Paul.Nom = "Paul";
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom);
    Console.Read();
}
```

Pas Paul / Pas Paul

Paul / Pas Paul

On peut constater que, dans le premier affichage, les deux variables "Paul" et "Lucie" contiennent les mêmes données. Après modification de la variable "Paul", on remarque que seule la variable "Paul" a été affectée par la modification, "Lucie" restant intact.

Les deux variables ainsi obtenues restent donc indépendantes.

3.1.3 Les énumérations

Les énumérations nous permettent d'établir des listes de choix auxquels est assignée une valeur. Cette liste ne peut pas être modifiée par le programme.

Ces types valeurs sont utilisés à des termes de compréhensibilité du code. En effet, il sera plus facilement compréhensible de mettre par exemple le nom d'un produit en vente plutôt que son identifiant numérique dans un code.

On peut aussi se servir des énumérations afin d'éviter aux développeurs travaillant avec nous de mettre une valeur numérique qui ne serait pas permise lors de l'appel d'une méthode par exemple.

```
'VB
Enum Metier As Integer
    Informaticien
    Jardinier
    Cuisinier
End Enum

//C#
enum Metier : int
{
    Informaticien,
    Jardinier,
    Cuisinier
}
```

Dotnet-France AS

3.2 Les variables de références

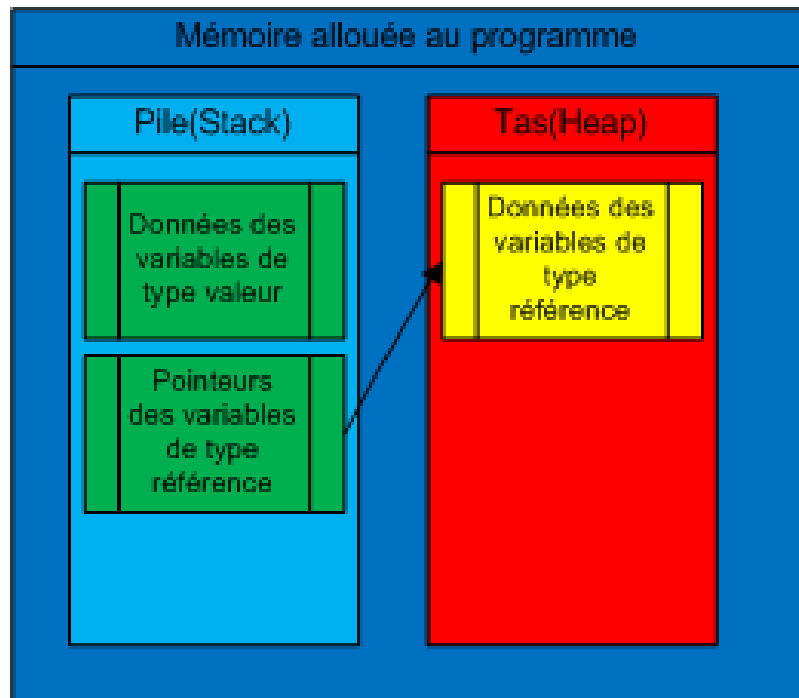
Les variables de références (ou type référence) fonctionnent de la même façon que les pointeurs dans le langage C : La variable ne contient plus directement une valeur mais une adresse en mémoire qui indique à quel endroit de la mémoire sont stockées les données.

Rappelez-vous:

- La mémoire d'un programme en .NET est coupée en deux: la pile et le tas.
- Les variables de type valeur n'utilisent que la pile.

Aussi on aurait pu se demander à quoi sert le tas mémoire. Vous avez la réponse dans les variables de type référence.

Le schéma ci-contre récapitule les types de variables et leur localisation en mémoire.



De telles variables sont créées lors de l'instanciation d'une classe pour obtenir un objet. Ainsi quand nous écrivons le code suivant:

```
'VB
Dim exemple As Object = New Object()

//C#
Object exemple = new Object();
```

Nous créons une variable de type référence nommée "exemple", stockée dans la pile, contenant un pointeur vers les données de type "Object" qui sont stockées dans le tas.

Note : Dans le cas de string, nous pouvons omettre le mot clef new, le compilateur se charge de remplacer l'affectation par sa version conforme utilisant le constructeur `new String`.

Il faut également savoir que le GarbageCollector n'agit que sur les données stockées dans le tas qui ne possèdent plus de référence dans la pile.

Contrairement aux variables de type valeur, nous manipulons donc un pointeur en mémoire. Nous allons reprendre le code de la [section 3.1.2](#) en remplaçant la structure "Personnage" par une classe du même nom:

```
'VB
Class Personnage
    Public Age As Integer
    Public Nom As String
    Public Sub New(ByVal nom As String, ByVal age As Integer)
        Me.Age = age
        Me.Nom = nom
    End Sub
    Public Overrides Function ToString() As String
        Return Nom + " = " + Age.ToString()
    End Function
    Public Shared Operator +(ByVal p1 As Personnage, ByVal p2 As
Personnage) As Personnage
        p1.Nom = p1.Nom + " et " + p2.Nom
        p1.Age = p1.Age + p2.Age
        Return p1
    End Operator
End Class
Sub Main()
    Dim Paul As Personnage = New Personnage("Pas Paul", 20)
    Dim Lucie As Personnage
    Lucie = Paul
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom)
    Paul.Nom = "Paul"
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom)
    Console.Read()
End Sub

//C#
class Personnage
{
    public int Age;
    public string Nom;
    public Personnage(string nom, int age)
    {
        this.Nom = nom;
        this.Age = age;
    }
    public override string ToString()
    {
        return Nom.ToString() + " = " + Age.ToString() + " ans";
    }
    public static Personnage operator +(Personnage p1, Personnage p2)
    {
        p1.Nom = p1.Nom + " et " + p2.Nom;
        p1.Age += p2.Age;
        return p1;
    }
}
public string main()
{
    Personnage Paul = new Personnage("Pas Paul", 20);
    Personnage Lucie;
    Lucie = Paul;
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom);
    Paul.Nom = "Paul";
    Console.WriteLine(Paul.Nom + " / " + Lucie.Nom);
    Console.Read();
}
```

Pas Paul / Pas Paul

Paul / Paul

Nous constatons cette fois-ci que si nousinstancions un objet Personnage, que nous copions la référence dans une autre variable référence et nous modifions le contenu pointé par la première variable référence, le contenu de la seconde variable change également.

La duplication de variables de type référence en utilisant l'opérateur "=" ne copie non plus les données de l'objet mais sa référence stockée dans la pile.

Grâce à ce procédé, toutes les recommandations de Microsoft concernant la taille des objets, leurs modifications après instanciation et leur changement de type par casting sont levées.

3.2.1 Quelques types références Built-In

Toutes les classes du Framework .NET donneront des variables de type référence. Il en existe plus de 2500 dans le Framework. Leur grand nombre est dû au fait que tout ce qui ne dérive pas de System.ValueType est de type référence.

Voici une petite liste de quelques types référence les plus utilisés:

Classe	Descriptif
System.String	Stocke un ensemble de caractère pour former des chaines de caractères de manière statique
System.Object	C'est la classe de base de toute classe dans le Framework .NET. Il n'y a pas de classe plus générale que celle-ci.
System.Text.StringBuilder	Effectue les mêmes opérations que String mais de manière dynamique
System.Array	Représente un tableau de données
System.IO.Stream	Offre un tampon de lecture et/ou d'écriture vers le réseau, un fichier ou un périphérique (Sera abordé dans le chapitre 2)
System.Exception	C'est la classe de base de toutes les erreurs pouvant être générées au cours de l'exécution du programme.

3.2.1.1 Les classes *String* et *StringBuilder*

Jusqu'à présent, nous avons utilisé nos types de façon simple, en se limitant à leur assigner une valeur et éventuellement à les additionner. Mais le fait d'utiliser des types qui dérivent de `System.Object` nous permet d'effectuer des traitements beaucoup plus complexes.

`String` et `StringBuilder` nous permettent d'effectuer des manipulations de chaînes de caractères de manière simplifiée. La différence majeure entre les deux est que `String` est non mutable, c'est-à-dire qu'à chaque traitement sur une chaîne de caractère, une nouvelle instance est créée et remplace la précédente, à l'inverse de `StringBuilder` qui est mutable et utilise un buffer pour effectuer les changements, sans créer de nouvelles instances de la chaîne en cours de traitement. `StringBuilder` est donc conseillé lorsqu'il s'agit de manipuler de nombreuses chaînes, `String` fera l'affaire pour de petits traitements et pour stocker une chaîne.

Voici comment instancier ces deux types référence :

```
'VB
Dim s1 As String = "bonjour"
s1 += "Ca va?"

Dim s2 As StringBuilder = New StringBuilder (30)
s2.Append( "Bonjour" )

//C#
String s1 = "Bonjour";
s1 += "Ca va?";

StringBuilder s2 = new StringBuilder(30);
s2.Append( "Bonjour" );
```

Les codes ci-dessus ont tous les mêmes effets: Ils déclarent et instancient une variable de type référence `String` et `StringBuilder` et ils concatènent la chaîne créée avec une autre.

3.2.1.2 La classe Array

Les arrays (traduisez Tableaux) se manipulent grâce aux membres de `System.Array`, soit avec les méthodes et propriétés statique de `System.Array`, soit avec les méthodes et propriétés propre à l'instance du tableau. Ils ne peuvent contenir qu'un seul type de données à la fois (un tableau d'entier, de flottants, de String, ...).

Dans l'exemple suivant, nous créons un tableau, nous le rangeons dans l'ordre numérique et nous l'invertissons. Puis on se contente d'afficher toutes les valeurs contenues.

```
'VB
Dim tableau() As Integer = {1, 5, 3, 4}
Array.Sort(tableau)
Array.Reverse(tableau)
Console.WriteLine("{0},{1},{2},{3}", tableau(0), tableau(1), tableau(2),
tableau(3))

//C#
int[] tableau = { 1, 5, 3, 4 };
Array.Sort(tableau);
Array.Reverse(tableau);
Console.WriteLine("{0},{1},{2},{3}", tableau[0], tableau[1], tableau[2],
tableau[3]);
```

5,4,3,1

3.2.1.3 La classe Exception

Les exceptions sont des événements remontés pendant l'exécution d'un assembly (votre exécutable ou une dll) en cas de problème.

Par exemple, dans le cas de notre concaténation de chaîne avec le type String, si la mémoire de la machine est saturée, une exception est levée.

Quand une exception est remontée, on a deux cas de figure qui s'offrent à nous:

- L'exception n'est pas gérée, le programme peut s'arrêter totalement car il ne peut pas traiter l'erreur générée (S'il n'y a plus de mémoire, il ne pourra pas aller plus loin)
- L'exception peut être gérée grâce à un bloc Try-Catch

Dans le cas du bloc Try-Catch (comprenez "Essais de...Si erreur, fait...") le programme va d'abord tenter d'exécuter le code contenu dans la clause Try et exécutera le code de la clause Catch uniquement si une erreur a été générée dans le bloc Try. Il est également possible de rajouter à un bloc Try-Catch une clause Finally qui sera toujours exécutée après le traitement de l'erreur ou s'il n'y a pas eu d'erreur.

Par exemple, vous cherchez à lire un fichier. Mais vous ne pouvez pas savoir si l'utilisateur du programme a créé ou non le fichier:

```
'VB
Try
    Dim sr As StreamReader = New StreamReader("Monfichier.txt")
Catch e As Exception
    Console.WriteLine("Une erreur est survenue : " + e.Message)
Finally
    Console.WriteLine("Retour au menu principal")
End Try

//C#
try
{
    StreamReader sr = new StreamReader("monfichier.txt");
}
catch (Exception e)
{
    Console.WriteLine("Une erreur est survenue : " + e.Message);
}
finally
{
    Console.WriteLine("Retour au menu principal");
}
}
```

```
Une erreur est survenue : Impossible de trouver le fichier 'C:\Users\Paul\Documents\Visual Studio
2008\Projects\Framework\Chapitre 1\bin\Debug\monfichier.txt'.
```

```
Retour au menu principal
```

Dans l'exemple ci-dessus, on va tenter d'ouvrir un fichier nommé "monfichier.txt". Si il y a une erreur, on affiche la ligne "Une erreur est survenue" et on utilise le message d'erreur contenu dans l'exception pour indiquer l'erreur précise. Dans tous les cas, on affichera la ligne "Retour au menu principal".

Le Framework .NET propose une centaine d'exception différente qui héritent toutes de la classe `System.Exception`. Il vous est également possible de créer vos exceptions personnalisées en créant des classes qui héritent de `System.Exception` (la notion d'héritage sera abordée plus bas).

Il faut également savoir que dans un bloc Try-Catch, plusieurs types d'exception différente peuvent être levés. Aussi, on peut tout à fait enchaîner plusieurs blocs Catch pour traiter tous les types d'erreur susceptible d'être levés durant l'exécution. Ceux-ci seront traités dans l'ordre (du premier au dernier) et la runtime exécutera le contenu du premier bloc Catch qui possède le même type d'exception qui vient d'être levée. Ce procédé est souvent appelé "Filtrage d'exception".

Généralement, lorsque nous réalisons un filtrage d'exception, nous filtrons les exceptions de la plus spécifique à la plus générale.

Note : Il existe deux types d'exception de bases appelées `ApplicationException` (pour spécifier une erreur générée par l'application) et `SystemException` (pour les erreurs liées au système). Ces deux types héritent d'`Exception`.

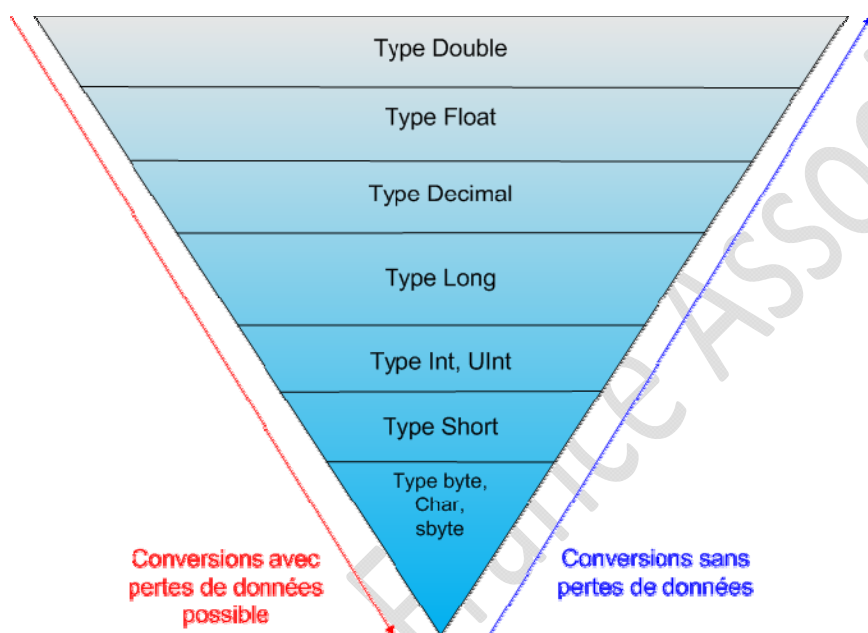
3.3 Les conversions de types

Les conversions de type sont utilisées couramment en .NET. Vous pouvez par exemple avoir besoin de convertir un type int en un type float pour effectuer des calculs de plus grande précision telle qu'une division.

3.3.1 Conversions de types

Il existe deux types de conversions :

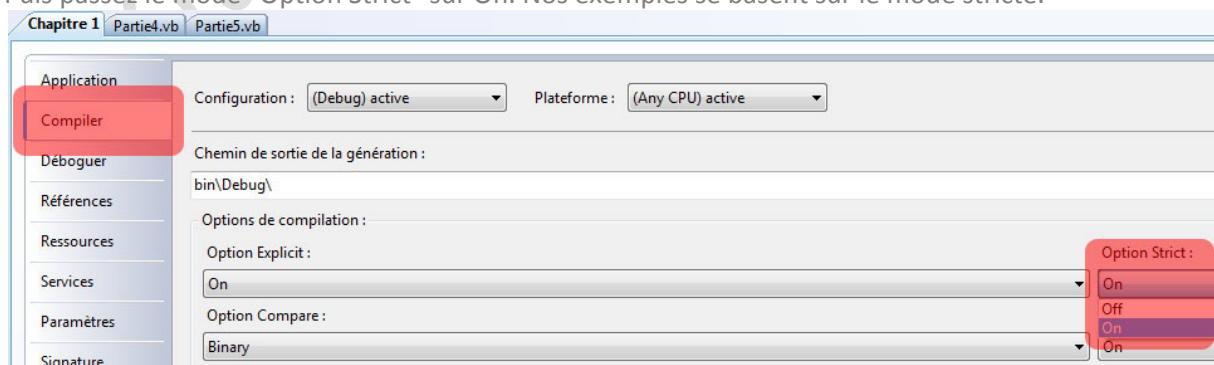
- Narrowing : Il s'agit de convertir un type d'un domaine étendu en un type possédant un domaine plus restreint (Double vers Int). Cette conversion a de fortes chances d'entraîner des pertes de données.
- Widening : Il s'agit de convertir un type d'un domaine restreint vers un type ayant un domaine plus étendu (Int vers Double). Cette conversion ne peut pas perdre de données.



Dans la [partie 3.1.1](#), nous évoquons quelques types valeurs disponibles dans le .NET Framework. Le schéma ci-contre indique les conversions possibles et leurs conséquences sur les données stockées. La flèche bleue indique donc les conversions de type Widening et la rouge indique les conversions de type Narrowing.

En Visual Basic, les conversions Narrowing et Widening sont implicites par défauts. En C#, les conversions Widening sont implicites alors que les conversions Narrowing doivent obligatoirement être explicitées !

Note : Il est possible de configurer Visual Basic afin qu'il ait le même comportement que le C# face aux conversions. Pour ceci, allez dans les propriétés du projet, sélectionnez l'onglet Compiler. Puis passez le mode "Option Strict" sur On. Nos exemples se basent sur le mode stricte.



Le .NET Framework nous fournit quelques outils pour effectuer ces conversions. Cependant, VB.NET et C# possèdent chacun des spécificités qui leur sont propres:

Méthode ou Classe	VB.NET	C#	Descriptif
System.Convert			Convertit les types qui implémentent l'interface System.IConvertible
	CType	Opérateur (type) pour effectuer un cast.	Convertit les types qui définissent des opérateurs de conversions explicites
type.ToString() type.Parse()			Convertit un objet en chaîne de caractères et inversement. Parse peut lever une exception.
type.TryParse() type.TryParseExact() (Exemple: System.Int32.TryParse())			Tente de convertir une chaîne de caractère en un type particulier. Retourne true si la conversion a réussi, false sinon.
	CBool, CInt, CStr ...		Offre de meilleure performance en VB.NET pour convertir un objet dans un type de base
	DirectCast TryCast		Effectue une conversion d'un type vers un autre. DirectCast peut lever une exception si les deux types n'ont aucune relation (héritage ou interface). TryCast se contentera de ne rien faire.

3.3.2 Le Boxing et l'Unboxing

Le Boxing, c'est le fait de convertir une variable de type valeur en une variable de type référence :

```
'VB
Dim i1 As Integer = 30
Dim o1 As Object = CType(i1, Object)

//C#
int i1 = 30;
Object o1 = (object)i1;
```

L'Unboxing effectue l'opération inverse (passer d'une variable de type référence a une variable de type valeur) :

```
'VB
Dim o2 As Object = 100
Dim i2 As Integer = CType(o2, Integer)

//C#
Object o2 = 100;
int i2 = (int)o2;
```



Important : Le Boxing et L'Unboxing consomment beaucoup de ressources lors de tâches répétitives. Il est possible d'optimiser les conversions entre types en utilisant le plus souvent possible ces bonnes pratiques :

- Si une de vos méthodes peut prendre plusieurs types différents en paramètre, il vaut mieux surcharger votre méthode autant de fois que nécessaire plutôt que d'utiliser une variable de type Object et de la convertir.
- Utilisez les types génériques plutôt que de passer des objets généraux en arguments.
- Surchargez les méthodes qui peuvent l'être (méthodes déclarées "virtual") fournis par System.Object plutôt que d'utiliser leur implémentation par défaut.

3.3.3 Implémentation des conversions dans les types personnalisés

L'implémentation des conversions dans vos types permet de simplifier la conversion de votre type vers un autre. Cela permet également, en cas de pertes de données éventuelles, d'effectuer des opérations supplémentaires.

Il existe plusieurs implémentations, nous vous en proposons trois couvrant la majorité des besoins :

- Pour les conversions Implicite / Explicite (Narrowing et Widening), il faut définir des opérateurs de conversions.
- Pour transformer un type quelconque en une chaîne de caractères, on surcharge la méthode ToString héritée implicitement de la classe System.Object.
- Implémenter l'interface System.IConvertible afin d'utiliser les méthodes de System.Convert.

3.3.3.1 Implémentation des conversions Implicites/Explicites

Cette implémentation permet de simplifier les conversions en définissant un comportement à un opérateur (par défaut, l'opérateur "="). Comme il existe des conversions implicite et explicite, nous allons définir le comportement de l'opérateur dans les deux cas grâce aux mots clef Implicit et Explicit (ou Widening et Narrowing en VB.NET).

```
//C#
struct Personnage
{
    public int age;

    public static implicit operator Personnage(int argument)
    {
        Personnage temp = new Personnage();
        temp.age = argument;
        return temp;
    }

    public static explicit operator int(Personnage argument)
    {
        return argument.age;
    }
}

static void Main(string[] args)
{
    Personnage v1;
    int v2;
    v1 = 50;
    v2 = (int) v1;

    Console.WriteLine("Valeur Personnage : {0} / Valeur int :
{1}", v1, v2);
}
```

```
'VB
Structure Personnage
    Implements IConvertible

    Public age As Integer

    Public Shared Widening Operator CType(ByVal argument As Integer)
As Personnage
    Dim temp As Personnage = New Personnage()
    temp.age = argument
    Return temp
End Operator

    Public Shared Narrowing Operator CType(ByVal argument As
Personnage) As Integer
    Return argument.age
End Operator
End Structure

Sub Main()
    Dim v1 As Personnage
    Dim v2 As Integer

    v1 = 50
    v2 = CType(v1, Integer)
    Console.WriteLine("Valeur Personnage : {0} / Valeur int : {1}", v1,
v2)
End Sub
```

L'exemple montre que dans le cas où l'on assigne une valeur à notre variable de Personnage, c'est la méthode implicite qui est utilisée. Celle-ci va créer une variable temporaire de Personnage et lui assigner la valeur en argument. La valeur en argument correspond à l'expression située à droite du signe égal.

Dans le cas où on souhaite assigner notre variable de type Personnage à une variable de type int, c'est la méthode explicite qui est utilisée. Pour ça, on explicite la conversion en effectuant un cast en C# ou en utilisant la méthode `CType` en VB.NET. Dans notre exemple, une telle conversion se contente de retourner la valeur "value" contenue dans notre type Personnage.

3.3.3.2 Convertir en chaîne de caractère avec ToString

L'implémentation par défaut de la méthode `ToString` dans la classe `Object` se contente de retourner le type de variable dont on appelle la méthode `ToString`.

Nous reprenons le code de la structure `Personnage` vu précédemment, et nous rajoutons le code suivant :

```
'VB
Structure Personnage
    Public Overrides Function ToString() As String
        Return Me.age.ToString()
    End Function
End Structure

Sub Main()
    Dim v1 As Personnage
    v1 = 50

    Console.WriteLine("Valeur Personnage : " + v1.ToString())
End Sub

//C#
struct Personnage
{
    public override string ToString()
    {
        return this.age.ToString();
    }
}

static void Main(string[] args)
{
    Personnage v1;
    v1 = 50 ;

    Console.WriteLine("Valeur Personnage : " + v1.ToString());
}
```

Nous affichons ainsi la valeur de la variable `Age` et non plus le type de la structure par défaut.

Dotnet

3.3.3.3 Implémentation de l'interface IConvertible

L'interface IConvertible permet quand elle est implémentée de définir le comportement des méthodes de System.Convert.

```
'VB
Structure Personnage
    Implements IConvertible

    Public age As Integer

    Public Function ToBoolean(ByVal provider As System.IFormatProvider)
As Boolean Implements System.IConvertible.ToBoolean
        If (Me.age <> 0) Then
            Return True
        Else
            Return False
        End If
    End Function
End Structure

Sub Main()
    Dim v1 As Personnage
    Dim v2 As Boolean
    v1 = 50
    v2 = Convert.ToBoolean(v1)
    Console.WriteLine("Valeur Personnage : {0} / Valeur v2 :
{1}",v1,v2)
End Sub
```

```
//C#
struct Personnage : IConvertible
{
    public int age;

    public bool ToBoolean(IFormatProvider provider)
    {
        if (this.age != 0)
            return true;
        else
            return false;
    }
}

static void Main(string[] args)
{
    Personnage v1;
    bool v2;

    v1 = 50;
    v2 = Convert.ToBoolean(v1);

    Console.WriteLine("Valeur Personnage : {0} / Valeur v2 : {1}",v1,v2);
}
```

Note : Vous n'êtes pas obligé de redéfinir le comportement de toutes les méthodes, il vous suffit d'envoyer une exception NotImplementedException pour chaque méthode non définie.

Note 2 : Vous verrez comment implémenter une interface page 29.

Dans ce code nous avons implémenté l'interface `IConvertible` et redéfinis la méthode `ToBoolean`. Nous avons volontairement masqué les autres méthodes compte tenu de leur nombre. Nous assignons la valeur 50 à notre variable `v1` et nous utilisons la méthode `ToBoolean` pour le convertir en booléen. Dans notre type nous indiquons que si `value` est différent de 0, alors il faut retourner vrai. Voici ce qui est affiché dans la console :

```
Valeur Personnage : 50 / Valeur v2 : True
```

Dotnet-France Association

4 Les particularités de la programmation orientée objet

La programmation orientée objet est un des piliers du Framework .NET. En effet, même pour la plus simple des applications, vous aurez toujours besoin à un moment d'utiliser une classe.

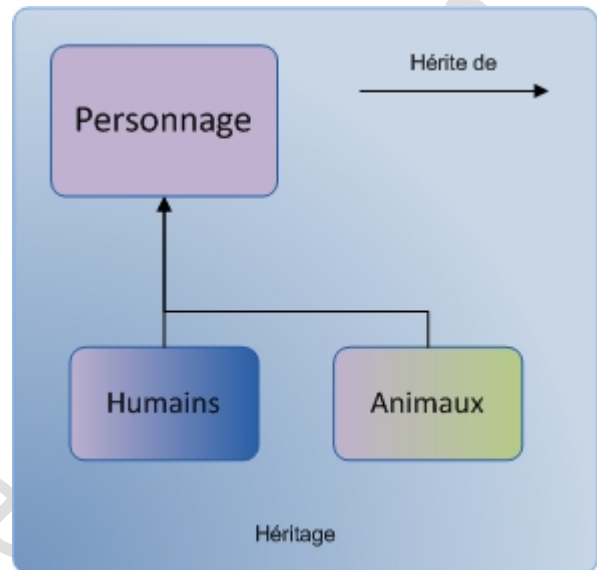
4.1 Héritage

Avant de commencer, nous tenons à vous rappeler une des notions fondamentale du Framework .NET : Toute classe dérive implicitement de la classe `System.Object`.

Cette hiérarchisation des classes se base sur une notion de la programmation orientée objet : l'héritage.

En faisant hériter une classe d'une autre, nous pouvons utiliser les méthodes, les propriétés et les attributs de la classe mère, et les spécificités de notre classe fille. Ainsi, toute classe créée peut avoir accès aux méthodes :

- `Equals`, indiquant si deux objets ont le même contenu.
- `GetHashCode` qui génère un hash à partir de la valeur de l'objet.
- `ToString` qui retourne une chaîne de caractères décrivant l'instance courante.



Prenons comme exemple une classe héritant de notre classe `Personnage` : nous avons écrit une classe représentant un personnage en général, mais nous souhaiterions différencier deux types de personnages : les Humains et les Animaux.

Nous allons donc faire hériter deux nouvelles classes (`Humains` et `Animaux`) de notre classe `Personnage`. Ainsi, nos objets `Humains` auront un attribut propre appelé `langue`, en plus des attributs d'un personnage et les objets `Animaux` auront un attribut `race` en plus des attributs d'un personnage.

Note : Il est impossible en C# et VB.NET de faire de l'héritage multiple. Vous ne pouvez donc pas faire hériter une classe de deux classes différentes.

```
'VB
Module Lesson3
    Public Class Humains : Inherits Lesson2.Personnage
        Public langue As String
    End Class

    Public Class Animaux : Inherits Lesson2.Personnage
        Public race As String
    End Class
End Module

//C#
class Humains : Personnage
{
    public string langue;
}

class Animaux : Personnage
{
    public string race;
}
```

Grâce au code ci-dessus, nous pouvons désormais instancier les classes Humains et Animaux qui auront respectivement les attributs "langue" et "race" en plus des attributs caractérisant un personnage à savoir "age" et "nom".

Note : Il est tout à fait possible de passer d'un objet "précis" (ici, un objet de type Humains ou Animaux) à un objet plus global (ici, un objet de type Personnage). Cette possibilité sera abordée plus tard.

4.2 Les interfaces

Une interface définit des méthodes et propriétés qui devront être impérativement implémentées dans la classe qui en hérite.

Les interfaces permettent de pallier au manque de l'héritage multiple, vous pouvez en effet implémenter d'autant d'interface que vous souhaitez. En contrepartie, vous devrez implémenter tous les membres contenus dans l'interface.

Note : Par convention le nom d'une interface commence par un i majuscule.

Afin d'illustrer cette explication par un exemple, nous allons créer une interface ICarac composée de deux membres : la méthode Aquatiques et la propriété Regime.

```
'VB
Public Interface ICarac
    Function Aquatiques() As Boolean
    Property Regime() As String
End Interface

//C#
interface ICarac
{
    bool Aquatiques();
    string Regime { get; set; }
}
```

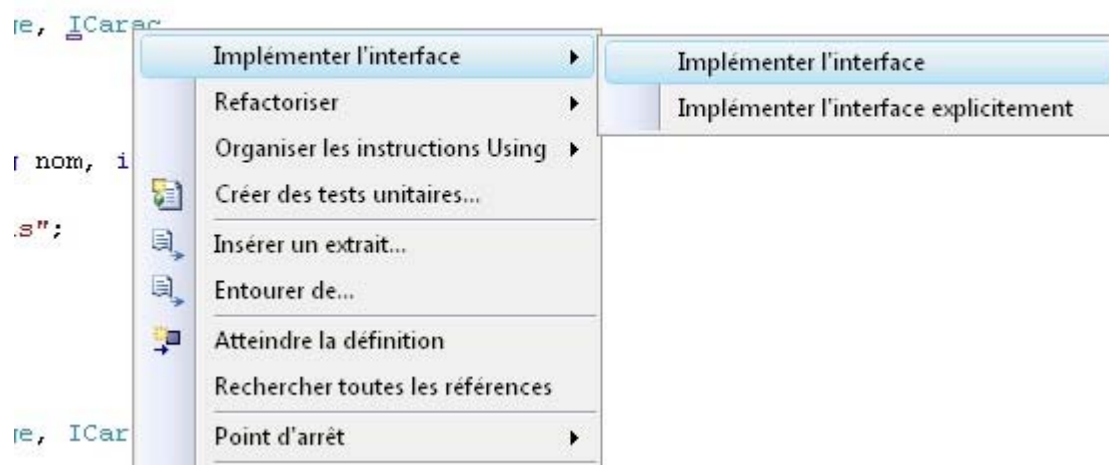
Pour implémenter l'interface dans nos classes, il faut d'abord hériter de l'interface :

```
'VB
Public Class Humains : Inherits partie32.Personnage
    Implements ICarac

//C#
class Humains : Personnage, ICarac
```

Ensuite, il y a deux méthodes différentes pour implémenter les propriétés et méthode d'interface :

- En C#, il suffit de cliquer avec le bouton droit sur ICarac puis cliquer sur "Implémenter l'interface → Implémenter l'interface" :



- En VB.NET, il suffit de placer le curseur après le nom de l'interface et d'appuyer sur la touche Entrée.

Il ne vous reste plus qu'à implémenter le comportement de chacune des propriétés et méthodes de l'interface.

Note : Vous pouvez créer une interface à partir d'une classe en utilisant Refactor. Cliquez avec le bouton droit sur le nom de votre classe, puis "Refactoriser → Extraire l'interface".

4.3 Les classes partielles

Le mot clef Partial vous permet de découper une classe dans plusieurs fichiers sources. L'intérêt principal est de découper une classe en plusieurs fichiers et, par conséquent, de découper le travail entre plusieurs développeurs sans se soucier des autres parties.

```
'VB
Partial Public Class MyClass

End Class

//C#
public partial class MyClass
{
}
```

Note : Le mot clef Partial a été ajouté dans le Framework 2.0.

Le développement d'applications WinForm utilise ce procédé pour:

- Stocker dans un fichier les codes de la classe principale, nécessaires à la réalisation de votre interface graphique créée avec le designer.
- Vous donner accès à la classe principale et son constructeur, sans pour autant surcharger le code de ladite classe (Vous remarquerez d'ailleurs que dans le constructeur, une méthode `InitializeComponent` est toujours placée par défaut. Cette méthode est implémentée dans l'autre fichier de code.)

4.4 Les classes génériques

Jusqu'ici, la solution pour faire transiter des données globales entre chaque classes et chaque méthodes était d'utiliser des objets de type Object. Cette méthode obligeait le développeur à effectuer de nombreux cast ce qui était à la fois une perte de performance logicielle (caster un objet d'un certain type dans un autre requiert beaucoup de ressources processeur) et une source importante d'erreur à l'exécution (par exemple, caster un objet de type String en un type Object avant de caster à nouveau l'objet de type Object en type Int n'empêchera pas le compilateur de compiler mais amènera inévitablement au crash de l'application).

Les types génériques permettent de créer des types de variables en laissant certaines parties inconnues, tout en conservant un typage fort (on dit que c'est une opération "type-safe"). Leur utilisation est très simple:

```
//C#
public class GenericClass<T, U>
{
    public T valeurA;
    public U valeurB;

    public GenericClass(T v1, U v2)
    {
        valeurA = v1;
        valeurB = v2;
    }
}
static void Main(string[] args)
{
    GenericClass<string, int> gen = new GenericClass<string, int>("Valeur
B vaut ", 35);
    Console.WriteLine(gen.valeurA + gen.valeurB.ToString());
}
```

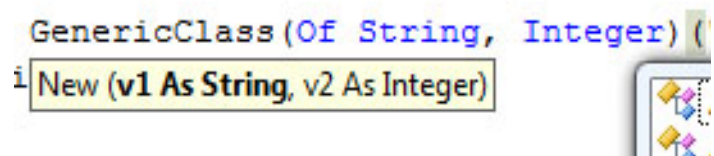
```
'VB
Public Class GenericClass(Of T, U)
    Public valeurA As T
    Public valeurB As U

    Public Sub New(ByVal v1 As T, ByVal v2 As U)
        valeurA = v1
        valeurB = v2
    End Sub
End Class
Sub Main()
    Dim gen As GenericClass(Of String, Integer) = New GenericClass(Of
String, Integer)("Valeur B vaut ", 35)
    Console.WriteLine(gen.valeurA + gen.valeurB.ToString())
End Sub
```

Valeur B vaut 35

Le code ci-dessus créé une classe utilisant deux types Générique appelés T et U.

Lors de ma déclaration dans la méthode Main, j'indique au compilateur que le type T sera utilisé en tant que type String et que le type U sera utilisé en tant que type Int. Au moment d'instancier mon objet, je peux d'ailleurs remarquer que l'IntelliSense a bien compris que mon objet utiliserait T en String et U en Int.



Je me contente ensuite d'afficher le String ainsi que la valeur entière contenue dans mon objet.

Les exemples d'utilisation des types générique sont les listes génériques contenues dans `System.Collections.Generic`. En effet, on ne peut pas prévoir quel type le développeur compte stocker dans une liste, aussi, nous laissons le type de données indéterminé.

Le plus souvent, on utilise deux types générique T et U. Il faut savoir que ces types génériques peuvent porter n'importe quel nom (qui ne soit pas déjà utilisé par un autre type) et que l'on peut très bien faire une classe générique avec autant de paramètres indéterminés qu'on le souhaite.



Important : Les types génériques étant non défini, on ne peut pas s'en servir à l'intérieur de la classe dans laquelle ils sont utilisés (ici dans la classe `GenericClass`). Cependant, nous savons que tout objet quel qu'il soit, hérite implicitement de la classe `System.Object`. Les seules méthodes disponibles sur les types génériques sont celle de la classe `Object`

Dotnet-France Association

4.5 Les évènements

Les évènements sont des actions déclenchées par des objets pour lancer l'exécution d'une portion de code. Par exemple, vous avez un bouton sur lequel l'utilisateur peut cliquer. Si il venait à cliquer sur ce bouton, un évènement sera généré et une portion de code sera exécutée (par exemple afficher un message de bienvenue).

Seulement, dans un tel procédé de communication, l'objet qui a généré l'évènement ne connaît pas la forme (type, nombres d'argument) de l'objet qui va recevoir l'évènement. Pour palier à ce problème, le Framework .NET met à disposition du développeur les types référence "delegate".

Les delegate sont en fait des classes qui ont la particularité de contenir non plus une référence vers des données mais une référence vers une méthode et possède donc une unique signature de cette méthode (ce sont les pointeurs de fonction en C).

```
'VB
Public Delegate Sub prototype(ByVal nom As String, ByVal valeur As Integer)

Public Class Delege
    Public Event evenement As prototype
    Public Sub genere()
        RaiseEvent evenement("Objet 1", 35)
    End Sub
End Class
Public Sub fonction_appellee(ByVal nom As String, ByVal valeur As Integer)
    Console.WriteLine(nom + " a la valeur " + valeur.ToString())
    Console.Read()
End Sub
Sub Main()
    Dim test As Delege = New Delege()
    AddHandler test.evenement, AddressOf fonction_appellee
    test.genere()
End Sub
```

```
//C#
public delegate void prototype(string nom, int valeur);

public class Delege
{
    public event prototype evenement;

    public void fonction_appellee(string nom, int valeur)
    {
        Console.WriteLine(nom + " a la valeur " + valeur.ToString());
    }
    public void genere()
    {
        evenement("Objet 1", 35);
    }
}

static void Main(string[] args)
{
    Delege test = new Delege();
    test.evenement += test.fonction_appellee;
    test.genere();
}
```

Ce code crée un nouveau délégué dont la signature est une fonction qui prend en argument une référence de type String et une valeur de type entière. Nous créons également une classe qui contient une méthode permettant de générer l'évènement et un attribut qui contiendra l'adresse vers la méthode à exécuter quand l'évènement est généré. La méthode appelée, nommée "fonction_appelee", se charge simplement d'afficher dans la console le String et le nombre passé en paramètres. Pour finir, la méthode main va créer un objet basé sur la classe créée, placer l'adresse de la méthode "fonction_appelee" dans la propriété "evennement" de l'objet et va appeler la méthode "genere" afin de générer l'évènement ce qui donne :

```
Objet 1 a la valeur 35
```

Dotnet-France Association

4.6 Les attributs

Les attributs permettent d'associer des informations ou un comportement à une classe, une méthode ou une propriété. On peut ensuite accéder dans le code à ces informations grâce à une méthode appelée "Réflexion".

Les attributs permettent par exemple d'indiquer les privilèges nécessaires pour que l'objet ciblé puisse s'exécuter, d'outrepasser certains privilèges afin d'obtenir une sécurité moins importante ou d'indiquer des capacités spéciales de l'objet.

L'exemple suivant se contente de créer une classe qui peut être sérialisée (c'est-à-dire que ses instances peuvent être enregistrées à un instant T dans un fichier):

```
'VB
<Serializable()> Public Class SerializableClass
End Class

//C#
[Serializable]
class SerializableClass
{ }
```

Les attributs disponibles dans le Framework .NET héritent tous de `System.Attribute`. Ainsi, il vous est tout à fait possible de créer vos attributs personnalisés en faisant hériter vos classes de `System.Attribute`.

4.7 Le TypeForwarding

Le TypeForwarding est un concept assez délicat à comprendre.

Prenons l'exemple de deux assemblies nommées A.dll et B.exe. Chacune de ces assemblies contient des codes.

Dans le développement du projet, on pourrait, à un moment donné, juger qu'une classe MyClass situé dans l'assembly A.dll aurait plutôt sa place dans l'assembly B.exe.

Pour cela, il n'y aura pas besoin de recréer tout le projet en changeant toute sa structure et ses espaces de noms, il suffira d'utiliser le TypeForwarding. Cela s'opère en quelques étapes:

- Couper la classe MyClass du fichier source contenant cette définition dans l'assembly A.dll.
- Coller le code de MyClass dans un fichier source de l'assembly B.exe et recompiler l'assembly B.exe.
- Ajouter quelques lignes de codes aux débuts des fichiers sources de l'assembly A.dll dans lesquels on utilisait la classe MyClass:

```
'VB
Imports System.Runtime.CompilerServices
Imports B
<Assembly: TypeForwardedTo(GetType(B.MyClass))>

//C#
using B;
using System.Runtime.CompilerServices;
[Assembly: TypeForwardedTo(GetType(B.MyClass))]
```

Pour donner un cas concret d'utilisation du TypeForwarding, mettez vous en situation:

Vous êtes le développeur d'un logiciel nommé "ComptaSoftware" et qui utilise deux librairies: Krypton Toolkit et Krypton Navigator, toutes deux développées par une même équipe que vous ne connaissez pas du tout. Vous utilisez dans votre application un contrôle "X" qui possède une classe du même nom et qui est actuellement placée dans la librairie Krypton Toolkit.

Seulement, au cours de l'évolution des outils Krypton, l'équipe de développement de ces librairies décide de déplacer le code du contrôle "X" de la librairie Krypton Toolkit vers la librairie Krypton Navigator.

Afin d'éviter aux développeurs comme vous, de devoir recompiler leur application pour prendre en compte les changements effectués dans les librairies, l'équipe de développement des outils Krypton va utiliser le TypeForwarding en indiquant dans la librairie Krypton Toolkit que la déclaration du contrôle "X" n'est plus dans Krypton Toolkit mais dans Krypton Navigator.

Ainsi, le contrôle "X" restera toujours accessible à partir de la librairie Krypton Toolkit même si la déclaration est réalisée dans la librairie Krypton Navigator.

Dotnet-France Association

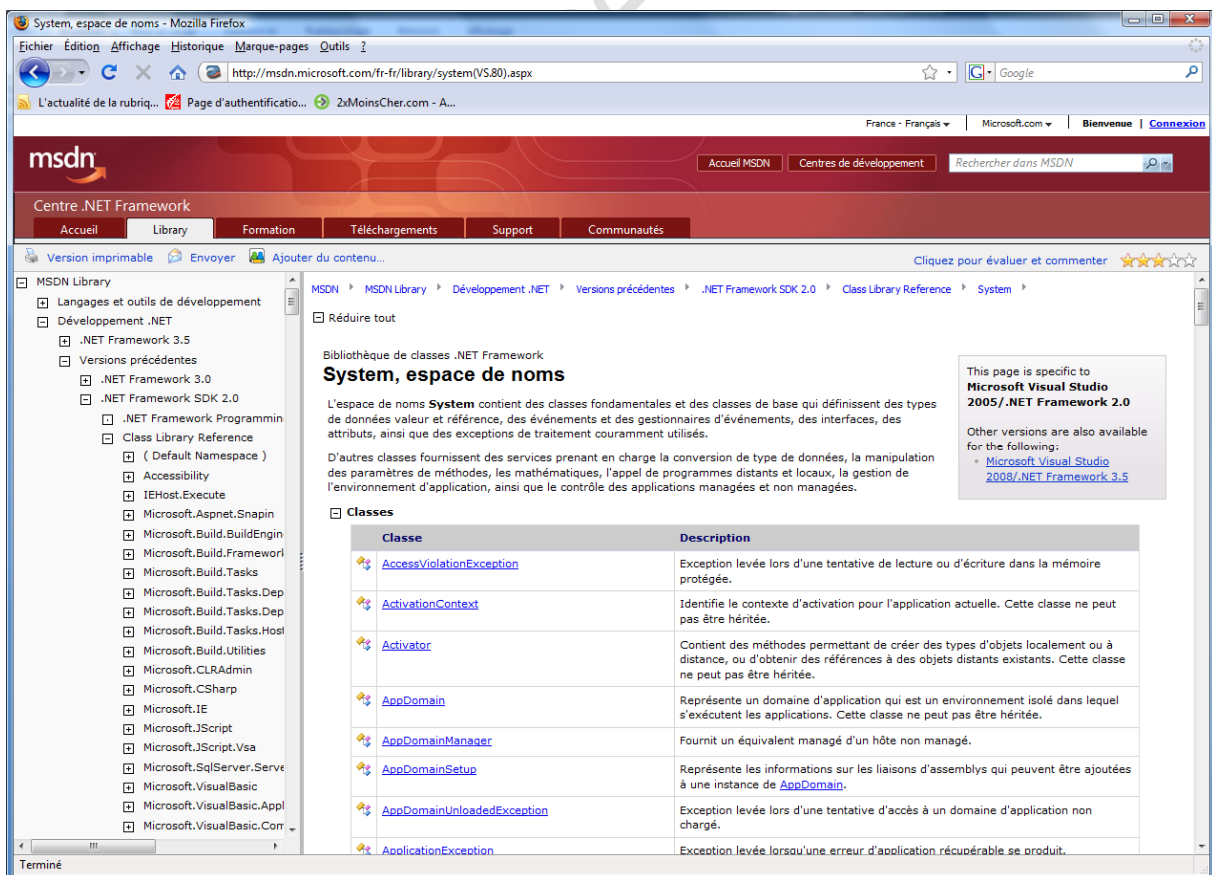
5 Conclusion

Dans ce premier chapitre, nous avons donc abordé quelles étaient les différences entre les variables de type valeur et les variables de type référence, comment les convertir entre-elles ou comment passer de l'une à l'autre et nous avons également vu comment créer ses types de variables personnalisés. Nous avons également abordé quelques points de la programmation orientée objet tels que la notion d'héritage et les interfaces.

Les points importants à retenir de ce module sont :

- Le découpage de la mémoire ainsi que la différence entre les types référence et les types valeur.
- La notion d'héritage et d'interface ainsi que ce qu'elles permettent de faire et comment elles s'implémentent.
- Que sont les événements, les classes partielles, les attributs, les classes génériques, les exceptions et avoir compris ce qu'ils permettent et comment ils s'implémentent.
- Comment convertir les types entre eux et comment passer d'un type valeur à un type référence.
- Connaître les types de conversions existantes et savoir ce qu'elles engendrent.
- Savoir implémenter de nouvelles définitions des opérateurs de conversions.
- Savoir ce que c'est que le GarbageCollector.

En outre, vous pouvez également vous aider du [MSDN](http://msdn.microsoft.com), site de référence pour tout ce qui concerne le développement ou l'utilisation des technologies Microsoft.



MSDN Library

Langages et outils de développement

Développement .NET

.NET Framework 3.5

Versions précédentes

.NET Framework 3.0

.NET Framework SDK 2.0

.NET Framework Programmation

Class Library Reference

(Default Namespace)

Accessibility

IEHost.Execute

Microsoft.AspNet.Snapin

Microsoft.Build.BuildEngine

Microsoft.Build.Framework

Microsoft.Build.Tasks

Microsoft.Build.Tasks.Dep

Microsoft.Build.Tasks.Host

Microsoft.Build.Utilities

Microsoft.CLRAdmin

Microsoft.CSharp

Microsoft.IE

Microsoft.JScript

Microsoft.JScript.Vsa

Microsoft.SqlServer.Serve

Microsoft.VisualBasic

Microsoft.VisualBasic.Appl

Microsoft.VisualBasic.Corr

MSDN > MSDN Library > Développement .NET > Versions précédentes > .NET Framework SDK 2.0 > Class Library Reference > System >

Réduire tout

Bibliothèque de classes .NET Framework

System, espace de noms

L'espace de noms **System** contient des classes fondamentales et des classes de base qui définissent des types de données valeur et référence, des événements et des gestionnaires d'événements, des interfaces, des attributs, ainsi que des exceptions de traitement couramment utilisés.

D'autres classes fournissent des services prenant en charge la conversion de type de données, la manipulation des paramètres de méthodes, les mathématiques, l'appel de programmes distants et locaux, la gestion de l'environnement d'application, ainsi que le contrôle des applications managées et non managées.

Classes

Classe	Description
AccessViolationException	Exception levée lors d'une tentative de lecture ou d'écriture dans la mémoire protégée.
ActivationContext	Identifie le contexte d'activation pour l'application actuelle. Cette classe ne peut pas être héritée.
Activator	Contient des méthodes permettant de créer des types d'objets localement ou à distance, ou d'obtenir des références à des objets distants existants. Cette classe ne peut pas être héritée.
AppDomain	Représente un domaine d'application qui est un environnement isolé dans lequel s'exécutent les applications. Cette classe ne peut pas être héritée.
AppDomainManager	Fournit un équivalent managé d'un hôte non managé.
AppDomainSetup	Représente les informations sur les liaisons d'assemblis qui peuvent être ajoutées à une instance de AppDomain .
AppDomainUnloadedException	Exception levée lors d'une tentative d'accès à un domaine d'application non chargé.
ApplicationException	Exception levée lorsqu'une erreur d'application récupérable se produit.

This page is specific to Microsoft Visual Studio 2005/.NET Framework 2.0

Other versions are also available for the following:

- [Microsoft Visual Studio 2008/.NET Framework 3.5](#)

Terminé