

Cours .NET avec C#

Programmation de base

Introduction

- C# est un langage orienté objet
 - Créé par Microsoft pour fonctionner avec la plateforme .NET
 - Utilisé pour créer des applications qui s'exécutent dans le CLR.NET
 - Permet de créer plusieurs types d'applications :
 - Application console exécutées à partir d'une ligne de commande
 - Applications Windows intégrant des contrôles (boutons, menus, ...)
 - Applications Web pouvant être visualisées à partir de n'importe quel navigateur Web
 - etc

Introduction

- Nous traitons C# d 'abord comme un langage de programmation classique.
- Nous aborderons les objets ultérieurement.
- La programmation classique s'efforce à séparer les données des instructions :
 - ❑ des données
 - ❑ les instructions qui les manipulent

Les données de C#

C# utilise les types de données suivants:

- les nombres entiers
- les nombres réels
- les caractères et chaînes de caractères
- les booléens
- les objets

Exemple d'une application Console

```
using System;  
class Program  
{  
    public static void Main(string[] args)  
    {  
        Console.Out.WriteLine("C'est un test");  
        Console.In.Read();  
    }  
}
```

Types simples

- En C# chaque variable doit être déclarée avant d'être utilisée
- La déclaration d'une variable se fait selon la syntaxe suivante :

Type_de données nom_variable[=valeur];

- C # fournit plusieurs types de données

Types simples : entiers

Type	Codage	Valeurs admises
sbyte	1 octet	-128 et 127
byte	1 octet	0 et 255
short	2 octets	-32768 et 32767
ushort	2 octets	0 et 65535
int	4 octets	-2147483648 et 2147483647
uint	4 octets	0 et 4294967296
long	8 octets	−9223372036854775808 et 9223372036854775807
ulong	8 octets	0 et 18446744073709551615

Types simples : réels

Type	Codage	Valeurs admises
float	4 octets	1.5×10^{-45} et 3.4×10^{38}
double	8 octets	5.0×10^{-324} et 1.7×10^{308}
decimal	16 octets	1.0×10^{-28} et 7.9×10^{28}

Types simples : autres

Type	Valeurs admises
char	Caractère
bool	Booléen (true ou false)
string	Chaîne de caractères

Types simples : entiers

Type	Alias
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64

Types simples : réels

Type	Alias
float	System.Single
double	System.Double
decimal	System.Decimal

Types simples : autres

Type	Alias
char	System.Char
bool	System.Boolean
string	System.String

Examples

- `int x=12;`
- `char a='A', b='b';`
- `String x="c:\\chap1\\paragraph3"`
- `String x=@"c:\chap1\paragraph3"`
- `bool v1=true, v2=false;`
- `Datetime date= new DateTime(1954,10,13);`

Boxing / Unboxing

Les opérations implicites de boxing/unboxing se font sur les types suivants :

int	Int32
long	Int64
decimal	Decimal
bool	Boolean
char	Char
byte	Byte
float	Float
double	Double
enum	Enum

Séquences d'échappement

- Chaîne ayant une signification particulière :
- \ ' : guillemet simple (quôte)
- \ » : guillemet
- \\ : barre oblique inverse
- \a : alerte
- \n : retour à la ligne
- \f : saut de page
- \t : tabulation horizontale
- \v : tabulation verticale

Conversion de types

En C#, on dispose de trois types de conversions :

- Conversion implicite
- Conversion explicite (cast)
- Conversion en utilisant des méthodes

Conversion implicite de types

On peut faire une conversion d'une manière implicite.

Exemples :

- ❑ `int a=6;` //le type int est codé sur 32 bits
- ❑ `long b=a;` //le type long est codé sur 64 bits
- ❑ Dans cet exemple, le sur-casting se fait implicitement

Conversion implicite de types

Conversion implicite est aussi appelé conversion élargissante (pas de perte d'information). Elle permet de passer de :

- ❑ byte : short, ushort, int, uint, long, ulong, float, double, decimal
- ❑ sbyte: short, int, long, float, double, decimal
- ❑ short: int, long, float, double, decimal
- ❑ ushort: int, uint, long, ulong, float, double, decimal
- ❑ int : long, float, double, decimal
- ❑ long : float, double, decimal
- ❑ ulong : float, double, decimal
- ❑ float : double
- ❑ char : ushort, int, uint, long, ulong, float, double, decimal

Conversion explicite de types

Un casting explicite peut être effectué simplement en faisant précéder la valeur par l'indication du nouveau type entre parenthèses.

Exemples :

```
double d=97.564;  
float f=(float)d;  
int i=(int)f;
```

Conversion explicite de types

- Cette conversion est appelée aussi conversion rétrécissante (perte d'informations)
 - ❑ byte: char
 - ❑ short: char, byte
 - ❑ char: short, byte
 - ❑ int: char, short, byte
 - ❑ long: int, char, short, byte
 - ❑ float: long, int, char, short, byte
 - ❑ double: float, long, int, char, short, byte

Conversion explicites à l'aide de Convert

- C# fournit des méthodes qui permettent de convertir des chaînes en nombres
- L'opération de conversion peut échouer dans le cas où la chaîne contient des données non numériques
- On trouve, les méthodes de la forme :
 - ❑ `Convert.ToXXX(chaine)`, Où XXX : indication du type destinataire
 - ❑ `Type.Parse(chaine)`, où type:int,Int32,long,Int64,double,Double,float,Float
- Ces méthodes font partie de l'espace de noms System

Conversion explicites à l'aide de Convert

Commande	Résultat
Convert.ToBoolean(val)	<i>val</i> convertie en <i>bool</i>
Convert.ToByte(val)	<i>val</i> convertie en <i>byte</i>
Convert.ToChar(val)	<i>val</i> convertie en <i>char</i>
Convert.ToDecimal(val)	<i>val</i> convertie en <i>decimal</i>
Convert.ToDouble(val)	<i>val</i> convertie en <i>double</i>
Convert.ToInt16(val)	<i>val</i> convertie en <i>short</i>
Convert.ToInt32(val)	<i>val</i> convertie en <i>int</i>

Conversion à l'aide des méthodes de Convert

Commande	Résultat
Convert.ToInt64(val)	<i>val</i> convertie en <i>long</i>
Convert.ToSByte(val)	<i>val</i> convertie en <i>sbyte</i>
Convert.ToSingle(val)	<i>val</i> convertie en <i>float</i>
Convert.ToString(val)	<i>val</i> convertie en <i>string</i>
Convert.ToUInt16(val)	<i>val</i> convertie en <i>ushort</i>
Convert.ToUInt32(val)	<i>val</i> convertie en <i>uint</i>
Convert.ToUInt64(val)	<i>val</i> convertie en <i>ulong</i>

Conversion l'aide des méthode de Parse

Commande	Résultat
<i>int.Parse(chaine)</i>	chaîne convertie en Int32
<i>Int32.Parse</i>	chaîne convertie en Int32
<i>long.Parse(chaine)</i>	chaîne convertie en Int64
<i>Int64.Parse</i>	chaîne convertie en Int64
<i>double.Parse(chaîne)</i>	chaîne convertie en Double
<i>Double.Parse(chaîne)</i>	chaîne convertie en Double
<i>float.Parse(chaîne)</i>	chaîne convertie en Float
<i>Float.Parse(chaîne)</i>	chaîne convertie en Float

Conversion l'aide des méthode de Parse

La conversion d'une chaîne vers un nombre peut échouer si la chaîne ne représente pas un nombre.

Il y a alors génération d'une erreur fatale appelée **exception** en C#.

Cette erreur peut être gérée par la clause *try/catch* :

```
try{  
    appel de la fonction susceptible de générer l'exception  
}catch (Exception e){  
    traiter l'exception e  
}  
instruction suivante
```

Opérateurs et expressions

- Une expression est une combinaison de variables et ou de constantes
- La combinaison se fait à l'aide d'opérateurs
- On distingue trois catégories d'opérateurs :
 - Unaires : appliqués à une seule opérande
 - Binaires : appliqués à deux opérandes
 - Ternaires : appliqués à trois opérandes

Opérateurs mathématiques

Opérateur	Catégorie	Utilisation
+	binaire	$v1 = v2 + v3$
-	Binaire	$v1 = v2 - v3$
*	Binaire	$v1 = v2 * v3$
/	Binaire	$v1 = v2 / v3$
%	Binaire	$v1 = v2 \% v3$

Opérateurs mathématiques

Opérateur	Catégorie	Utilisation
+	Unaire	$v1 = + v3$
-	Unaire	$v1 = - v3$
++	Unaire	$v1 = ++v3,$ $v1 = v3++$
--	Unaire	$v1 = --v3$ $v1 == v3--$

Opérateurs d'affectation

Opérateur	Catégorie	Utilisation
=	Binaire	$v=w$
+=	Binaire	$v+=w$ ($v=v+w$)
-=	Binaire	$v-=w$ ($v=v-w$)
=	Binaire	$v=w$ ($v=v*w$)
/=	Binaire	$v/=w$ ($v=v/w$)
%=	Binaire	$v\%=w$ ($v=v\%w$)

Opérateurs relationnels

Opérateur	Catégorie	Utilisation
<code>==</code>	Binaire	$v == w$
<code>!=</code>	Binaire	$v != w$
<code><</code>	Binaire	$v < w$
<code><=</code>	Binaire	$v \leq w$
<code>></code>	Binaire	$v > w$
<code>>=</code>	binaire	$v \geq w$

Opérateurs logiques

Opérateur	Catégorie	Utilisation
!	Unaire	var1=!var2
&&	Binaire	var1=var2&&var3
	Binaire	var1=var2 var3
^	binaire	var1=var2^var3

Opérateurs par priorité

Opérateurs unaires	++exp,--exp,+exp,-exp, !
Multiplicatifs	*, /, %
Additifs	+, -
Relationnels	<, <=, >, >=
Égalité	==, !=
Et logique	&&
Ou logique	
affectation	=, +=, -=, *=, /=, %=

Opérateurs

- Les opérateurs les plus prioritaires sont évalués en premier
- Deux opérateurs de même priorité sont évalués de gauche à droite, à l'exception des opérateurs d'affectation qui sont évalués de droite à gauche

Lecture/écriture

La classe `System.Console` donne accès aux opérations de lectures / écriture :

- **`System.Console.Out.Write()`** affiche
- **`System.Console.Out.WriteLine()`** affiche et passe à la ligne suivante
- **`System.Console.In.Read()`** lit un caractère et retourne son code
- **`System.Console.In.ReadLine()`** lit une ligne et retourne une chaîne de caractères

Lecture/écriture

Remarque :

Au lieu de donner la spécification complète des méthodes de lecture écriture, on peut utiliser l'espace des noms **System**, ainsi :

```
System.Console.Out.WriteLine();
```

Est équivalent à

```
Using System;
```

```
...
```

```
Console.Out.WriteLine()
```

Structures de contrôle

- L'instruction conditionnelle if

```
if (expression) instruction;
```

ou :

```
if (expression) {  
    instruction1;  
    instruction2;  
}
```

- L'instruction conditionnelle else

```
if (expression) {  
    instruction1;  
}else {  
    instruction2;  
}
```

Structures de contrôle

■ Les instructions conditionnelles imbriquées

```
if (expression1) {  
    bloc1;  
}  
else if (expression2) {  
    bloc2;  
}  
else if (expression3) {  
    bloc3;  
}  
else {  
    bloc5;  
}
```

Structures de contrôle

■ L'instruction switch

```
switch( variable) {  
    case valeur1: instructions1;break;  
    case valeur2: instructions2;break;  
    case valeur3: instructions2;break;  
    .  
    .  
    case valeurN: instructionsN;break;  
    default: instructions;break;  
}
```

Structures de contrôle

■ La boucle for

La boucle **for** est une structure employée pour exécuter un bloc d'instructions un nombre de fois en principe connu à l'avance. Elle utilise la syntaxe suivante :

```
for (initialisation;test;incrémentation) {  
instructions;  
}
```

Exemple :

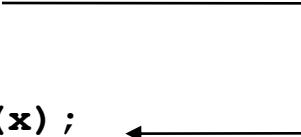
```
int i = 0;  
for (i = 2; i < 10;i++) {  
System.out.println("Vive Java !");  
}
```

Structures de contrôle

- *Branchement au moyen des instructions `break` et `continue`*


break:

```
int x = 10;  
for (int i = 0; i < 10; i++) {  
    x--;  
    if (x == 5) break; _____  
}  
System.out.println(x); ← _____
```

A horizontal line extends from the end of the 'if' statement, then turns vertically down and then horizontally left, ending with an arrow pointing to the 'println' statement, indicating that the loop is exited.

continue:

```
for (int i = 0; i < 10; i++) { ← _____  
    if (i == 5) continue; _____  
    System.out.println(i);  
}
```

A horizontal line extends from the end of the 'if' statement, then turns vertically up and then horizontally left, ending with an arrow pointing to the 'for' loop header, indicating that the rest of the loop body is skipped and the next iteration begins.

Structures de contrôle

■ L'instruction While

```
while (condition){  
    BlocInstructions;  
}
```

Exemple :

```
int s=0,i=0;  
while (i<10){  
    s+=i;  
    i++;  
}
```

■ L'instruction do .. while

```
do  
    BlocInstructions;  
while (condition)
```

Exemple :

```
int s=0,i=0;  
do{  
    s+=i;  
    i++;  
}  
while (i<10)
```