Fondamentaux C#

7 JUILLET

Auteur : Aurélien BOUDIER

Types et expressions primitives

Variables et constantes

Une variable est un nom que nous donnons à un emplacement de stockage en mémoire. Nous utilisons des variables pour stocker des valeurs temporaires en mémoire.

Une constante est une valeur qui ne peut pas être modifiée. Nous utilisons des constantes dans les situations où nous devons nous assurer qu'une valeur ne change pas. Par exemple, si vous travaillez sur un programme de calcul de l'aire d'un cercle, vous devez utiliser le nombre Pi (3.14). Vous pouvez définir Pi comme une constante pour vous assurer de ne pas modifier accidentellement la valeur de Pi dans les calculs.

Pour définir une variable, nous spécifions un type et un identifiant :

```
int number;
```

Ici, int représente le type entier, qui prend 4 octets en mémoire.

number est l'identifiant de notre variable. Nous pouvons éventuellement définir la valeur d'une variable sur la déclaration. Cela s'appelle « initialiser une variable » :

```
int number = 5;
```

Nota bene : en C #, vous ne pouvez pas lire la valeur d'une variable à moins que vous ne l'ayez définie auparavant.

Types primitifs

Type	Bytes
byte	1
short	2
int	4
long	8
float	4
double	8
decimal	16
bool	1
char	2

Ces types ont un type équivalent dans .NET Framework. Ainsi, lorsque vous compilez votre application, le compilateur mappe vos types au type sous-jacent dans .NET Framework.

C# Type	.NET Type
byte	Byte
short	Int16
int	Int32
long	Int64
float	Single
double	Double
decimal	Decimal
bool	Boolean
char	Char

Le moyen facile de mémoriser les types Int * est de se souvenir du nombre d'octets que chaque type utilise.

Par exemple, un "short" prend 2 octets. Nous avons 8 bits dans chaque octet. Donc, un "short" prend 16 octets en mémoire, par conséquent, le type .NET sous-jacent est Int16.

A noter qu'il existe aussi quelques types non-primitifs (string, array, struct, class, etc...)

Scope

La scope (la portée) détermine l'endroit où une variable a un sens et est accessible. Une variable a une portée dans le bloc dans lequel elle est définie, mais aussi dans tous les blocs enfants. Mais elle n'est pas accessible en dehors de ce bloc. Un bloc est indiqué par des accolades ({}).

Plage des types intégraux

C#	Plage	Type .NET
sbyte	-128 à 127	System.SByte
byte	0 à 255	System.Byte
short	-32 768 à 32 767	System.Int16
ushort	0 à 65 535	System.UInt16
int	-2 147 483 648 à 2 147 483 647	System.Int32
uint	0 à 4 294 967 295	System.UInt32
long	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	System.Int64
ulong	0 à 18 446 744 073 709 551 615	System.UInt64

Overflowing

Chaque type de variable peut stocker une plage de valeurs (avec une valeur minimum et une valeur maximum) comme nous venons de le voir. Si nous stockons une valeur dans une variable et que cette valeur dépasse la limite des valeurs pour le type sous-jacent, un débordement (overflowing) se produit. Par exemple, nous pouvons stocker toutes les valeurs comprises entre 0 et 255 dans un octet. Si la valeur d'un octet dépasse cette limite pendant les calculs, un débordement se produit.

Voici un exemple :

```
byte b = 255;

b = b + 1;
```

À la suite de la deuxième ligne, la valeur de b sera 0.

Il faudra être vigilant lorsque nous manipulerons des nombres. Il existe des méthodes qui génèrent des exceptions en cas d'overflowing.

Conversion de type

Il y a des moments où vous devez convertir temporairement la valeur d'une variable en un type différent.

Notez que cette conversion n'a pas d'impact sur la variable d'origine car C# est un langage à type statique, ce qui signifie en termes simples : une fois que vous déclarez le type d'une variable, vous ne pouvez pas le changer. Mais vous devrez peut-être convertir la valeur d'une variable dans le cadre de l'attribution de cette valeur à une variable d'un type différent.

Il existe quelques scénarios de conversion :

Si les types sont compatibles (par exemple, les nombres entiers et les nombres réels) et que le type cible est plus grand (plus grand scope de valeurs), vous n'avez rien à faire. La valeur sera automatiquement convertie par le CLR et stockée dans le type cible.

Exemple :

```
byte b = 1;

int i = b;
```

Ici, b est de type byte et n'utilise donc qu'un seul octet en mémoire, nous pouvons facilement le convertir en int qui occupera 4 octets en mémoire. Nous n'avons donc rien à faire.

Cependant, si le type cible est plus petit que le type source, le compilateur générera une erreur.

La raison en est qu'un débordement peut se produire dans le cadre de la conversion. Par exemple, si vous avez un int avec la valeur 1000, vous ne pouvez pas le stocker dans un byte car la valeur maximale d'un byte de 255. Dans ce cas, certains bits seront perdus en mémoire. Et c'est la raison pour laquelle le compilateur vous avertira de ces scénarios. Si vous êtes sûr qu'aucun bit ne sera perdu lors de la conversion, vous pouvez dire au compilateur que vous êtes conscient du débordement et que vous souhaitez toujours que la conversion se produise. Dans ce cas, vous utiliserez un cast :

```
int i = 1;

octet b = (octet) i;
```

Dans cet exemple, notre int contient la valeur 1, qui peut parfaitement être stockée dans un byte. Nous pouvons utiliser un cast pour dire au compilateur d'ignorer le débordement. Un cast signifie « préfixer la variable » du même type que le type cible. Donc, ici, nous convertissons la variable i en byte dans la deuxième ligne.

Enfin, si les types source et cible ne sont pas compatibles (par exemple une chaîne et un nombre), vous devez utiliser la classe Convert (native au C#).

```
string str = "1234";

int i = Convert.ToInt32 (str);
```

La classe Convert a un certain nombre de méthodes pour convertir des valeurs en différents types.

Les opérateurs

En C#, nous avons 4 types d'opérateurs :

• Arithmétiques : utilisés pour les calculs

• Comparaisons : utilisés pour comparer les valeurs dans les expressions booléennes

• Logiques : représentent un ET logique, un OU ou un NON

• Bitwise : operateurs binaires

Opérateurs arithmétiques :

Operateur	Description
+	Ajouter
-	Soustraire
*	Multiplier
/	Diviser
%	Reste de la division (modulo)
++	Incrémenter de 1
--	Décrémenter de 1

Opérateurs de comparaison :

Operateur	Description
>	Plus grand
>=	Plus grand ou égal
<	Inferieur
<=	Inferieur ou égal
==	Est égal a
!=	Diffèrent de

Opérateurs logiques :

Operateur	Description
&&	ET logique
||	OU logique
!	NON logique

Opérateurs binaires :

Operateur	Description
&	ET binaire
|	OU binaire

Crédit :

Action	Auteur	Descriptif	Date
Réalisation du document	Aurélien BOUDIER	Fondamentaux du C#	07/07/20