C# : variables, constantes et structures conditionnelles

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Les variables
   - Les opérations de lecture et écriture
   - Les variables explicitement typées
   - Les variables implicitement typées
2. Les opérations sur les variables
3. Les structures conditionnelles
4. Les constantes
5. Les mots-clés `out`, `ref`, `in` et `params`
6. La conversion

C#

Une variable ?

- Une zone mémoire
- Permettant de stocker une ou plusieurs données
- Pouvant avoir plusieurs valeurs différentes dans un programme

C#

Une variable ?

- Une zone mémoire
- Permettant de stocker une ou plusieurs données
- Pouvant avoir plusieurs valeurs différentes dans un programme

C# est un langage de programmation fortement typé

- Il faut préciser le type de chaque variable
- Une variable peut changer de valeurs mais ne peut jamais changer de type.

C#

Pourquoi typer les variables ?

Pour

- connaître l'espace de stockage nécessaire pour la variable.
- déterminer des valeurs minimale et maximale pour la variable.
- pouvoir appliquer des méthodes et des opérations sur les valeurs de ce type.

C#

Pourquoi typer les variables ?

Pour

- connaître l'espace de stockage nécessaire pour la variable.
- déterminer des valeurs minimale et maximale pour la variable.
- pouvoir appliquer des méthodes et des opérations sur les valeurs de ce type.

En C#, il y a deux types de variable

- variables explicitement typés : type précisé à la déclaration
- variables implicitement typés : type déterminé selon la valeur affecté à la variable à la déclaration

C#

Pour lire une chaîne saisie dans la console et l'enregistrer dans une variable

```
string s = Console.ReadLine();
```

`string` est le type de la valeur saisie et sauvegardée dans la variable `s`

C#

Pour lire une chaîne saisie dans la console et l'enregistrer dans une variable

```
string s = Console.ReadLine();
```

`string` est le type de la valeur saisie et sauvegardée dans la variable `s`

Pour afficher le contenu d'une variable dans la console

```
Console.Write("chaine saisie : {0}",s);
```

{0} fait référence à la première variable située après le texte du message à afficher.

C#

Pour lire une chaîne saisie dans la console et l'enregistrer dans une variable

```
string s = Console.ReadLine();
```

`string` est le type de la valeur saisie et sauvegardée dans la variable `s`

Pour afficher le contenu d'une variable dans la console

```
Console.Write("chaine saisie : {0}",s);
```

{0} fait référence à la première variable située après le texte du message à afficher.

On peut aussi utiliser la syntaxe suivante pour l'affichage d'une variable

```
Console.Write($"chaine saisie : {s}");
```

$ permet de remplacer les variables situées entre { } par leurs valeurs respectives

C#

Déclarer une variable

```
type nomVariable;
```

C#

Principaux types simples en c# (le nom d'un type commence par une lettre en minuscule)

- `sbyte` : entier codé sur 1 octet, valeur comprise entre -128 et 127 (équivalent non signé `byte`, valeur comprise entre 0 et 255)
- `short` : entier codé sur 2 octets (entre -32 768 et 32 767, équivalent non signé `ushort`)
- `int` : entier codé sur 4 octets (entre 2 147 483 648 et 2 147 483 647, équivalent non signé `uint`)
- `long` : entier codé sur 8 octets (entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 807, équivalent non signé `ulong`)
- `float` : nombre à virgule codé sur 4 octets
- `double` : nombre à virgule codé sur 8 octets
- `decimal` : nombre à virgule codé sur 16 octets
- `bool` : variable booléenne acceptant les valeurs `true` ou `false`
- `char` : caractère codé sur 2 octet situé entre deux `'`
- `string` : une chaîne de caractère situé entre deux `"`

C#

Exemple

```
int x;
```

C#

Exemple

```
int x;
```

Déclarer et initialiser une variable

```
int x = 5;
```

C#

Exemple

```
int x;
```

Déclarer et initialiser une variable

```
int x = 5;
```

Ceci est une erreur

```
byte x = 300;
```

C#

Exemple

```
int x;
```

Déclarer et initialiser une variable

```
int x = 5;
```

Ceci est une erreur

```
byte x = 300;
```

Pour convertir le contenu d'une variable (le `cast` pour les types compatibles)

```
int x = 100;
byte z = (byte) x;
Console.Write(z); // affiche 100
```

C#

Attention aux valeurs qui dépassent l'intervalle

```
int x = 300;
byte z = (byte) x;
Console.WriteLine(z); // affiche 44
```

C#

Attention aux valeurs qui dépassent l'intervalle

```
int x = 300;
byte z = (byte) x;
Console.WriteLine(z); // affiche 44
```

Pour connaître le type d'une variable

```
int x = 300;
Console.WriteLine(x.GetType());
// affiche System.Int32
```

C#

Attention aux valeurs qui dépassent l'intervalle

```
int x = 300;
byte z = (byte) x;
Console.WriteLine(z); // affiche 44
```

Pour connaître le type d'une variable

```
int x = 300;
Console.WriteLine(x.GetType());
// affiche System.Int32
```

`int` VS `Int32`

- `int` est un synonyme (raccourci) de `Int32`
- En programmation, `int` est plus familier que `Int32`

C#

La liste des synonymes

- `Byte` pour `byte`
- `SByte` pour `sbyte`
- `Int16` pour `short`
- `Int64` pour `long`
- `Int32` pour `int`
- `Single` pour `float`
- `Double` pour `double`
- `Boolean` pour `bool`
- `Char` pour `char`
- `String` pour `string`

C#

Le type `Nullable`

- Les variables de type numérique, les variables booléennes et les caractères n'acceptent pas la valeur `null`
- Pour chacun de ces types non `nullable`, il existe un type `nullable` qui lui est associé `type?`

C#

Le type `Nullable`

- Les variables de type numérique, les variables booléennes et les caractères n'acceptent pas la valeur `null`
- Pour chacun de ces types non `nullable`, il existe un type `nullable` qui lui est associé `type?`

Pour les entiers

```
int? x = null;
Console.WriteLine(x); // affiche une ligne vide
x = 7;
Console.WriteLine(x); // affiche 7
```

C#

Pour tester si x a une valeur

```
int? x = null;
if (x.HasValue)
{
  Console.WriteLine($"La valeur de x est {x}");
}
else
{
  Console.WriteLine("x n'a pas de valeur");
}
```

C#

Pour tester si x a une valeur

```
int? x = null;
if (x.HasValue)
{
  Console.WriteLine($"La valeur de x est {x}");
}
else
{
  Console.WriteLine("x n'a pas de valeur");
}
```

Modifier la valeur de x (2 par exemple) et tester

C#

Conversion d'un type `Nullable`

```
int? x = null;
int y = x ?? 5;
Console.WriteLine($"La valeur de y est  {y}");
```

C#

Conversion d'un type `Nullable`

```
int? x = null;
int y = x ?? 5;
Console.WriteLine($"La valeur de y est  {y}");
```

Modifier la valeur de y (2 par exemple) et tester

C#

Conversion d'un type `Nullable`

```
int? x = null;
int y = x ?? 5;
Console.WriteLine($"La valeur de y est  {y}");
```

Modifier la valeur de y (2 par exemple) et tester **On peut aussi faire la**

conversion explicite

```
int? x = null;
int y = (int)x;
Console.WriteLine($"La valeur de y est  {y}");
```

C#

Conversion d'un type `Nullable`

```
int? x = null;
int y = x ?? 5;
Console.WriteLine($"La valeur de y est  {y}");
```

Modifier la valeur de y (2 par exemple) et tester **On peut aussi faire la**

conversion explicite

```
int? x = null;
int y = (int)x;
Console.WriteLine($"La valeur de y est  {y}");
```

Attention une exception sera levée si y est nulle

C#

Le type `bool?`

- peut contenir trois valeurs différentes : `true`, `false` et `null`
- Les opérateurs logiques possibles sont & (et) et | (ou)

C#

Le type `bool?`

- peut contenir trois valeurs différentes : `true`, `false` et `null`
- Les opérateurs logiques possibles sont & (et) et | (ou)

Le résultat des opérations logiques quand la valeur `null` est présente

a	b	a & b	a | b
`true`	`null`	`null`	`true`
`false`	`null`	`false`	`null`
`null`	`null`	`null`	`null`

C#

Les variables implicitement typées

- Les variables implicitement typées sont déclarées avec le mot clé `var` sans préciser le type
- Il faut initialiser la valeur de la variable implicitement typée à la déclaration
- une fois la variable initialisée, la valeur aura le type de la valeur et ne peut donc plus changer de valeur

C#

Déclaration + initialisation d'une variable avec le mot clé `var`

```
var x = 2;
```

C#

Déclaration + initialisation d'une variable avec le mot clé `var`

```
var x = 2;
```

La variable x aura comme type `int` (`Int32`)

```
x = 2;
Console.WriteLine(x.GetType());
```

C#

Déclaration + initialisation d'une variable avec le mot clé `var`

```
var x = 2;
```

La variable x aura comme type `int` (`Int32`)

```
x = 2;
Console.WriteLine(x.GetType());
```

Ceci est une erreur

```
x = 2;
Console.WriteLine(x.GetType());
x = "bonjour";
Console.WriteLine(x.GetType());
```

C#

Exemple d'affectation de type

```
var x = 2;
Console.WriteLine(x.GetType());
// affiche Int32

var y = 2L;
Console.WriteLine(y.GetType());
// affiche Int64

var z = 2f;
Console.WriteLine(z.GetType());
// affiche Single

var t = 2.0;
Console.WriteLine(t.GetType());
// affiche Double
```

C#

Pour les variables numériques (`int`, `float`...)

- `=` : affectation
- `+` : addition
- `-` : soustraction
- `*` : multiplication
- `/` : division
- `%` : reste de la division

C#

Exemple

```
int x = 5;
int y = 2;
Console.WriteLine($"{x + y}"); // affiche 7
Console.WriteLine($"{x - y}"); // affiche 3
Console.WriteLine($"{x * y}"); // affiche 10
Console.WriteLine($"{x / y}"); // affiche 2
Console.WriteLine($"{(float)x / y}"); // affiche 2,5
Console.WriteLine($"{x % y}"); // affiche 1
```

C#

Quelques raccourcis

- `i = i + 1` ⇒ `i++;`
- `i = i - 1` ⇒ `i--;`
- `i = i + 2` ⇒ `i+=2;`
- `i = i - 2` ⇒ `i-=2;`

C#

Exemple de post-incrémentation

```
int i = 2;
int j = i++;
Console.WriteLine(i); // affiche 3
Console.WriteLine(j); // affiche 2
```

C#

Exemple de post-incrémentation

```
int i = 2;
int j = i++;
Console.WriteLine(i); // affiche 3
Console.WriteLine(j); // affiche 2
```

Exemple de pre-incrémentation

```
int i = 2;
int j = ++i;
Console.WriteLine(i); // affiche 3
Console.WriteLine(j); // affiche 3
```

C#

L'opérateur + pour concaténer deux chaînes de caractère

```
String str1 = "bon";
String str2 = "jour";
Console.WriteLine(str1 + str2);
// affiche bonjour
```

C#

Quelques méthodes pour les chaînes de caractères

- `Length()` : retourne le nombre de caractère de la chaîne.
- `IndexOf(x)` : retourne l'indice de la première occurrence de la valeur de `x` dans la chaîne, -1 sinon.
- `Contains(x)` : retourne `true` si la chaîne contient la sous-chaîne `x`, `false` sinon.
- `Substring(i,j)` : permet d'extraire une sous-chaîne de taille `j` de la chaîne à partir de l'indice `i`
- `Equals(str)` : permet de comparer la chaîne à `str` et retourne `true` en cas d'égalité, `false` sinon.
- `Replace(old,new)` : permet de remplacer toute occurrence de la chaîne `old` dans la chaîne courante par `new` et retourne la nouvelle chaîne
- ...

Pour accéder à un caractère d'indice `i` d'une chaîne `str`, il faut écrire `str[i]`. Le premier caractère est d'indice 0.

C#

Exemple : `Replace(old,new)`

```
String str = "bonjour";
Console.WriteLine($"{str.Replace("jour","soir")} ");
// affiche bonsoir
```

C#

Exemple : `Replace(old,new)`

```
String str = "bonjour";
Console.WriteLine($"{str.Replace("jour","soir")} ");
// affiche bonsoir
```

Exemple : `IndexOf(str,startIndex)`

```
String str = "bonjour les bons jours";
int pos = str.IndexOf("bon", 5);
Console.WriteLine($"{pos} ");
// affiche 12
```

C#

Tester une condition

```
if (condition1) {
    ...
}
[
else if (condition2) {
    ...
}
...
else {
    ...
}
]
```

C#

Opérateurs logiques

- && : et
- || : ou
- ! : non

C#

Opérateurs logiques

- && : et
- || : ou
- ! : non

Tester plusieurs conditions (en utilisant des opérateurs logiques)

```
if (condition1 && !condition2 || condition3){
    ...
}
[else ...]
```

C#

Opérateurs logiques

- `&&` : et
- `||` : ou
- `!` : non

Tester plusieurs conditions (en utilisant des opérateurs logiques)

```
if (condition1 && !condition2 || condition3){
    ...
}
[else ...]
```

Pour les conditions, on utilise des opérateurs de comparaisons

C#

Opérateurs de comparaison

- == : pour tester l'égalité
- != : pour tester l'inégalité
- > : supérieur à
- < : inférieur à
- >= : supérieur ou égal à
- <= : inférieur ou égal à

C#

Opérateurs de comparaison

- == : pour tester l'égalité
- != : pour tester l'inégalité
- > : supérieur à
- < : inférieur à
- >= : supérieur ou égal à
- <= : inférieur ou égal à

En C#, on ne peut comparer deux valeurs de type incompatible

C#

Structure conditionnelle avec `switch`

```
int x = 5;
switch (x) {
    case 1:
      Console.WriteLine("un");
      break;
    case 2:
      Console.WriteLine("deux");
      break;
    case 3:
      Console.WriteLine("trois");
      break;
    default:
      Console.WriteLine("autre");
      break;
}
```

C#

La variable dans `switch` peut être de type

- numérique : `int`, `long`...
- booléen : `bool`
- text : `char` ou `string`
- énumération

C#

Le bloc `default` dans `switch`

- Le bloc `default` peut apparaître à n'importe quelle position dans `switch`. Quelle que soit sa position, il est toujours évalué en dernier, une fois que tous les blocs `case` ont été évaluées.
- En l'absence d'un bloc `default` et si aucun bloc `case` n'est exécuté, le bloc `switch` sera traversé sans être exécuté.
- `break` permet de quitter `switch`
- Même dans bloc `default`, il faut placer un `break`.

C#

On peut aussi simplifier les tests en utilisant les expressions ternaires (Elvis Operator)

```
int x = 2;
String  type = ( x % 2 == 0 ) ? "pair" : "impair";
Console.WriteLine(type); // affiche pair
```

C#

Une constante ?

- c'est un élément qui ne peut changé de valeur

C#

Une constante ?

- c'est un élément qui ne peut changé de valeur

Pour déclarer une constante

- il faut utiliser le mot-clé `const`

C#

Une constante ?

- c'est un élément qui ne peut changé de valeur

Pour déclarer une constante

- il faut utiliser le mot-clé `const`

Déclaration d'une constante

```
const double pi = 3.1415;
```

C#

Une constante ?

- c'est un élément qui ne peut changé de valeur

Pour déclarer une constante

- il faut utiliser le mot-clé `const`

Déclaration d'une constante

```
const double pi = 3.1415;
```

L'instruction suivante ne peut être acceptée

```
pi = 5;
```

C#

Définitions

- `ref` : permet à une méthode d'utiliser et modifier la copie originelle de la valeur d'une variable passée en paramètre
- `out` : oblige une méthode à modifier la valeur d'une variable passée en paramètre
- `in` (depuis C# 7.2) : permet à une méthode d'utiliser la copie originelle de la valeur d'une variable passée en paramètre mais sans pouvoir la modifier

C#

Considérons la méthode suivante qui permet d'échanger les valeurs de deux variables entières

```
public static void Permutation(int i, int j)
{
    int aux = i;
    i = j;
    j = aux;
}
```

C#

Considérons la méthode suivante qui permet d'échanger les valeurs de deux variables entières

```
public static void Permutation(int i, int j)
{
    int aux = i;
    i = j;
    j = aux;
}
```

Et si on teste cette méthode en ajoutant le code suivant dans le `Main`

```
int n = 2;
int m = 5;
Permutation(n, m);
Console.WriteLine($"Après permutation, n = { n }");
// affiche Après permutation, n = 2
Console.WriteLine($"Après permutation, m = { m }");
// affiche Après permutation, m = 5
```

C#

Que s'est-il passé ?

- Par défaut, les types simples sont passés par valeur
- C'est à dire la méthode appelée (ici `Permutation`) travaille seulement sur une copie de la variable
- La méthode appelante (ici `Main`) conserve donc la valeur originelle de la variable

C#

Que s'est-il passé ?

- Par défaut, les types simples sont passés par valeur
- C'est à dire la méthode appelée (ici `Permutation`) travaille seulement sur une copie de la variable
- La méthode appelante (ici `Main`) conserve donc la valeur originelle de la variable

Comment faire pour travailler sur la valeur originelle ?

- Il faut effectuer un passage par référence

C#

Il faut ajouter le mot-clé `ref` dans la signature de la méthode appelée

```
public static void Permutation(ref int i, ref int j)
{
    int aux = i;
    i = j;
    j = aux;
}
```

C#

Il faut ajouter le mot-clé `ref` dans la signature de la méthode appelée

```
public static void Permutation(ref int i, ref int j)
{
    int aux = i;
    i = j;
    j = aux;
}
```

Et aussi lors de l'appel de cette méthode dans `Main`

```
int n = 2;
int m = 5;
Permutation(ref n, ref m);
Console.WriteLine($"Après permutation, n = { n }");
// affiche Après permutation, n = 5
Console.WriteLine($"Après permutation, m = { m }");
// affiche Après permutation, m = 2
```

C#

Considérons deux méthodes qui permettent de retourner la valeur `min` ou `max`

```
public static int FindMax(int i, int j)
{
    return i > j ? i : j;
}

public static int FindMin(int i, int j)
{
    return i < j ? i : j;
}
```

C#

Considérons deux méthodes qui permettent de retourner la valeur `min` ou `max`

```
public static int FindMax(int i, int j)
{
    return i > j ? i : j;
}

public static int FindMin(int i, int j)
{
    return i < j ? i : j;
}
```

Comment faire pour fusionner les deux méthodes ?

Sachant qu'une méthode (tout comme une fonction) ne peut retourner qu'une seule valeur

C#

Il faut ajouter les valeurs à retourner comme paramètres de la méthode et les précéder par `out`

```
public static void FindMinMax(int i, int j, out int max, out
   int min)
{
    max = i > j ? i : j;
    min = i < j ? i : j;
}
```

C#

Il faut ajouter les valeurs à retourner comme paramètres de la méthode et les précéder par `out`

```
public static void FindMinMax(int i, int j, out int max, out
   int min)
{
    max = i > j ? i : j;
    min = i < j ? i : j;
}
```

Pour appeler cette méthode dans le `Main`

```
int x;
int y;
FindMinMax(2, 3, out x, out y);
Console.WriteLine($"Le max de 2 et 3 est : { x }");
// affiche 3
Console.WriteLine($"Le min de 2 et 3 est : { y }");
// affiche 2
```

C#

Remarque

- Contrairement à `ref`, les paramètres précédés par `out` peuvent ne pas être initialisés

C#

Hypothèse

- Si on veut modifier la méthode `FindMax` pour qu'elle retourne la valeur maximale quel que soit le nombre de paramètres passés

C#

Hypothèse

- Si on veut modifier la méthode `FindMax` pour qu'elle retourne la valeur maximale quel que soit le nombre de paramètres passés

Solution

- utiliser l'argument `params`

C#

La méthode FindMaxFromNValue

```
public static int FindMaxFromNValue(params int[] list)
{
    int max = list[0];
    for (int i = 1; i < list.Length; i++)
    {
        if (list[i] > max)
            max = list[i];
    }
    return max;
}
```

C#

La méthode FindMaxFromNValue

```
public static int FindMaxFromNValue(params int[] list)
{
    int max = list[0];
    for (int i = 1; i < list.Length; i++)
    {
        if (list[i] > max)
            max = list[i];
    }
    return max;
}
```

Pour appeler cette méthode dans le Main avec un nombre de paramètres différents

```
int h = FindMaxFromNValue(2, 8, 5);
Console.WriteLine($"Le max est : { h }");

int g = FindMaxFromNValue(1, 7, 8, 2);
Console.WriteLine($"Le max est : { g }");
```

C#

Pour lire un caractère saisi dans la console

```
char c = Console.ReadKey().KeyChar;
Console.WriteLine("caractere saisi : {0}", c);
```

C#

Pour lire un caractère saisi dans la console

```
char c = Console.ReadKey().KeyChar;
Console.WriteLine("caractere saisi : {0}", c);
```

ou aussi

```
char c = (char)Console.Read();
Console.WriteLine("caractere saisi : {0}", c);
```

C#

Pour lire un caractère saisi dans la console

```
char c = Console.ReadKey().KeyChar;
Console.WriteLine("caractere saisi : {0}", c);
```

ou aussi

```
char c = (char)Console.Read();
Console.WriteLine("caractere saisi : {0}", c);
```

Pour attendre la saisie d'un caractère sans le récupérer

```
Console.ReadKey();
```

C#

Ceci ne permet pas de lire un chiffre

```
int j = Console.Read();
Console.WriteLine("chiffre saisi : {0}", j);
```

ça affiche son code ASCII

C#

Ceci ne permet pas de lire un chiffre

```
int j = Console.Read();
Console.WriteLine("chiffre saisi : {0}", j);
```

ça affiche son code ASCII

Pour lire un entier saisi dans la console

il faut

- lire une chaîne de caractère
- ensuite la convertir

C#

Lire une chaîne

```
string s =  Console.ReadLine();
```

Convertir la saisie : première méthode

```
int j = int.Parse(s);
Console.WriteLine("entier saisi : {0}", j);
```

Si `s` contient du texte, une exception de type `FormatException` sera levée

C#

Lire une chaîne

```
string s =  Console.ReadLine();
```

Convertir la saisie : première méthode

```
int j = int.Parse(s);
Console.WriteLine("entier saisi : {0}", j);
```

Si `s` contient du texte, une exception de type `FormatException` sera levée

Convertir la saisie : deuxième méthode

```
int k = Convert.ToInt16(s);
Console.WriteLine("entier saisi : {0}", k);
```

Si `s` contient du texte, une exception de type `FormatException` sera aussi levée

C#

Convertir la saisie : troisième méthode

```
int l;
int.TryParse(s, out l);
Console.WriteLine("entier saisi : {0}", l);
```

Si `s` contient du texte, aucune exception ne sera levée et `l` contiendra 0

C#

Remarque

- `int.TryParse` retourne `true` si la conversion a eu lieu, `false` sinon.
- En utilisant `Convert`, il faut préciser le nombre de bits pour coder l'entier.