Introduction à la programmation en C#

Alexandre Meslé

13 mars 2013

Ce document n'est un modeste support de cours. Il est mis à jour au fur et à mesure que je prépare mes cours, et toute remarque permettant de l'améliorer est, bien entendu, la bienvenue.

Il s'adresse à des personnes n'ayant jamais programmé, et donc reprend toutes les bases depuis le début. Il s'adresse toutefois à un public de futurs professionnels, par conséquent il est assez dense et comporte de nombreux détails qui sont souvent omis dans les tutoriaux.

Comme je me forme à ce langage en même temps que je rédige ce support, j'utilise à la fois le site du zéro et mes propres exercices pour apprendre ce langage de programmation. Je vous invite, si vous souhaitez progresser, à consulter ce tutoriel parallèlement au mien, vous y trouverez les mêmes concepts expliqués différemment. Et je vous invite aussi faire les exercices que je propose, ils sont disposés par ordre de difficulté croissante et intégralement corrigés.

Bon courage !

Table des matières

1 Notes de cours **5**

1.1 Introduction . . . 5
1.1.1 Définitions et terminologie . . . 5
1.1.2 Hello World ! . . . 6
1.1.3 Quelques explications . . . 6
1.2 Variables . . . 8
1.2.1 Déclaration . . . 8
1.2.2 Affectation . . . 8
1.2.3 Saisie . . . 9
1.2.4 Affichage . . . 9
1.2.5 Entiers . . . 9
1.2.6 Nombre décimaux à point-fixe . . . 10
1.2.7 Flottants . . . 10
1.2.8 Caractères . . . 11
1.2.9 Chaînes de caractères . . . 11
1.3 Opérateurs . . . 12
1.3.1 Généralités . . . 12
1.3.2 Les opérateurs unaires . . . 12
1.3.3 Les opérateurs binaires . . . 13
1.3.4 Formes contractées . . . 14
1.3.5 Opérations hétérogènes . . . 15
1.3.6 Les priorités . . . 16
1.4 Traitements conditionnels . . . 17
1.4.1 Si ... Alors . . . 17
1.4.2 Switch . . . 20
1.4.3 Booléens . . . 21
1.4.4 Les priorités . . . 22
1.5 Boucles . . . 23
1.5.1 Définitions et terminologie . . . 23
1.5.2 **while** . . . 23
1.5.3 **do** ... **while** . . . 24
1.5.4 **for** . . . 24
1.5.5 Accolades superflues . . . 25
1.6 Chaînes de caractères . . . 26
1.6.1 Exemple . . . 26
1.6.2 Définition . . . 26
1.6.3 Déclaration et initialisation . . . 26
1.6.4 Opérations . . . 26
1.6.5 Chaînes modifiables . . . 27
1.7 Tableaux . . . 29
1.7.1 Définitions . . . 29
1.7.2 Déclaration . . . 29
1.7.3 Initialisation . . . 30

1.7.4 Accès aux éléments . . . 30
1.7.5 Exemple . . . 30
1.7.6 L'instruction **foreach** . . . 32
1.8 Sous-programmes . . . 33
1.8.1 Les procédures . . . 33
1.8.2 Variables locales . . . 35
1.8.3 Passage de paramètres . . . 36
1.8.4 Les fonctions . . . 38
1.8.5 Passages de paramètre par référence . . . 40
1.9 Objets . . . 42
1.9.1 Création d'un type . . . 42
1.9.2 L'instanciation . . . 42
1.9.3 Les méthodes . . . 43
1.9.4 Le mot-clé **this** . . . 46

2 Exercices **48**
2.1 Variables . . . 48
2.1.1 Saisie et affichage . . . 48
2.1.2 Entiers . . . 49
2.1.3 Flottants . . . 49
2.1.4 Caractères . . . 49
2.2 Opérateurs . . . 50
2.2.1 Conversions . . . 50
2.2.2 Opérations sur les bits (difficiles) . . . 50
2.2.3 Morceaux choisis (difficiles) . . . 51
2.3 Traitements conditionnels . . . 52
2.3.1 Prise en main . . . 52
2.3.2 Switch . . . 52
2.3.3 L'échiquier . . . 53
2.3.4 Heures et dates . . . 53
2.3.5 Intervalles et rectangles . . . 54
2.4 Boucles . . . 55
2.4.1 Compréhension . . . 55
2.4.2 Utilisation de toutes les boucles . . . 56
2.4.3 Choix de la boucle la plus appropriée . . . 56
2.4.4 Morceaux choisis . . . 57
2.4.5 Extension de la calculatrice . . . 58
2.4.6 Révisions (SISR) . . . 58
2.5 Chaînes de caractères . . . 59
2.5.1 Prise en main . . . 59
2.5.2 Morceaux choisis . . . 59
2.6 Tableaux . . . 60
2.6.1 Exercices de compréhension . . . 60
2.6.2 Prise en main . . . 60
2.6.3 Indices . . . 61
2.6.4 Recherche séquentielle . . . 61
2.6.5 Morceaux choisis . . . 61
2.7 Sous-programmes . . . 63
2.7.1 Géométrie . . . 63
2.7.2 Arithmétique . . . 65
2.7.3 Passage de tableaux en paramètre . . . 66
2.7.4 Décomposition en facteurs premiers . . . 67
2.8 Objets . . . 69
2.8.1 Création d'une classe . . . 69
2.8.2 Méthodes . . . 69

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Définitions et terminologie

Un programme **exécutable** est une suite d'instructions exécutée par le processeur. Ces instructions sont très difficile à comprendre, si par exemple, vous ouvrez avec un éditeur de texte (notepad, emacs, etc) un fichier exécutable, vous verrez s'afficher un charabia incompréhensible :

```
00000000
0000001f
0000003e
0000005d
0000007c
0000009b
000000ba
000000d9
000000f8
00000117
00000136
00000155
00000174
00000193
000001b2
000001d1
000001f0
0000020f
0000022e
0000024d
0000026c
0000028b
000002aa
000002c9
000002e8
00000307
...
```

Il n'est pas envisageable de créer des programmes en écrivant des suites de chiffres et de lettres. Nous allons donc utiliser des langages de programmation pour ce faire.

Un programme C# est un **ensemble d'instructions** qui se saisit dans un fichier `.cs` à l'aide d'un

éditeur, ce type de fichier s'appelle une **source**. Les instructions qui y sont écrites s'appellent du **code** ou encore le **code source**. Un compilateur est un logiciel qui lit le code source et le convertit en un **code exécutable**, c'est-à-dire un ensemble d'instructions compréhensible par le processeur.
Certains environnement de développement servent d'éditeur, et prennent en charge la compilation et l'exécution (eclipse, Microsoft Visual C# 2010 Express, monodevelop, sharpdevelop...).

1.1.2 Hello World !

Allez sur `http://msdn.microsoft.com/fr-fr/express/aa975050`, téléchargez et installez le framework c#. Ouvrez un nouveau projet en mode cohsole nommé `HelloWorld`, et copiez-collez le code ci-dessous dans l'éditeur :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Dans la barre d'outils cliquez sur la flèche verte et vous pourrez voir votre fichier s'exécuter.

Pour que la console soit visible lors de l'exécution faites : outils → personnaliser → Commandes → Ajouter une commande → Déboguer → Exécuter sans débogage, puis cliquez sur la flèche verte qui a été ajoutée lors de cette manipulation.

1.1.3 Quelques explications

Corps du programme

```
static void Main(string[] args)
{
        Console.WriteLine("Hello World!");
}
```

On place dans les accolades du main les instructions que l'on souhaite voir s'exécuter :

```
static void Main(string[] args)
{
        <instructionsAExecuter>
}
```

Remarquez que chaque ligne se termine par un point-virgule. Pour afficher une variable en C#, on utilise `Console.WriteLine("message a afficher");`. Les valeurs entre doubles quotes sont affichées telles quelles.

Commentaires

Un commentaire est une séquence de caractères ignorée par le compilateur, on s'en sert pour expliquer des portions de code. Alors ayez une pensée pour les pauvres programmeurs qui vont reprendre votre code après vous, le pauvre correcteur qui va essayer de comprendre votre pensée profonde, ou bien plus

simplement à vous-même au bout de six mois, quand vous aurez complètement oublié ce que vous aviez dans la tête en codant. On délimite un commentaire par `/*` et `*/`. Par exemple,

```
static void Main(string[] args)
{
        /*
        Ceci est un commentaire :
                L'instruction ci-dessous affiche ''Hello world!''
        Ces phrases sont ignorées par le compilateur.
        */
        console.WriteLine("Hello world!");
}
```

1.2 Variables

Une variable est un emplacement de la mémoire dans lequel est stockée une valeur. Chaque variable porte une nom et c'est ce nom qui sert à identifier l'emplacement de la mémoire représenté par cette variable. Pour utiliser une variable, la première étape est la déclaration.

1.2.1 Déclaration

Déclarer une variable, c'est prévenir le compilateur qu'un nom va être utilisé pour désigner un emplacement de la mémoire. En C#, on déclare les variables à l'intérieur du bloc formé par les accolades du `Main`. Il faut toujours déclarer les variables avant de s'en servir.

Nous ne travaillerons pour le moment que sur les variables de type numérique entier. Le type qui y correspond, en C# est **int**. On déclare les variables entières de la manière suivante :

```
public static void Main(string[] args)
{
        int variable1, variable2, ..., variablen;
        ...
}
```

Cette instruction déclare les variables `variable1`, `variable2`, ..., `variablen` de type entier. Par exemple,

```
int variable1, variable2;
int autrevariable1, autrevariable2;
```

1.2.2 Affectation

Si on souhaite affecter à la variable `v` une valeur, on utilise l'opérateur =. Par exemple,

```
int v;
v = 5;
```

Cet extrait de code déclare une variable de type entier que l'on appelle `v`, puis lui affecte la valeur 5. Comme vous venez de le voir, il est possible d'écrire directement dans le code une valeur que l'on donne à une variable.

Il est aussi possible d'initialiser une variable en même temps qu'on la déclare. Par exemple, l'extrait ci-dessus se reformule

```
int v = 5;
```

Les opérations arihmétiques disponibles sont l'addition (+), la soustraction (−), la multiplication (∗), la division entière dans l'ensemble des entiers relatifs (quotient : /, reste : %). Par exemple,

```
int v, w, z;
v = 5;
w = v + 1;
z = v + w / 2;
v = z % 3;
v = v * 2;
```

1.2.3 Saisie

Traduisons en C# l'instruction `Saisir <variable>` que nous avons vu en algorithmique. Pour récupérer la saisie d'un utilisateur et la placer dans une variable `<variable>`, on utilise l'instruction suivante :

```
<variable> = int.Parse(Console.ReadLine());
```

Ne vous laissez pas impressionner par l'apparente complexité de cette instruction, elle suspend l'exécution du programme jusqu'à ce que l'utilisateur ait saisi une valeur et pressé la touche **return**. La valeur saisie est alors affetée à la variable `<variable>`. Par exemple, pour déclarer une variable `i` et l'initialiser avec une saisie, on procède comme suit :

```
int i = int.Parse(Console.ReadLine());
```

1.2.4 Affichage

Traduisons maintenant l'instruction `Afficher <variable>` en C# :

```
Console.WriteLine(<variable>);
```

Cette instruction affiche la valeur contenue dans la variable `variable`. Nous avons étendu, en algorithmique, l'instruction `Afficher` en intercalant des valeurs de variables entre les messages affichés. Il est possible de faire de même en C# :

```
Console.WriteLine("la valeur de la variable v est " + v + ".");
```

Les valeurs ou variables affichées sont ici séparés par des +. Tout ce qui est délimité par des double quotes est affiché tel quel. Cette syntaxe s'étend à volonté :

```
Console.WriteLine("les valeurs des variables x, y et z sont " + x +
", " + y + " et " + z);
```

1.2.5 Entiers

Quatre types servent à représenter les entiers :

nom	taille (t)	nombre de valeurs (2^{8t})
byte	1 octet	2^8 valeurs
short	2 octet	2^{16} valeurs
int	4 octets	2^{32} valeurs
long	8 octets	2^{64} valeurs

Plages de valeurs

Il nécessaire en programmation de représenter des valeurs avec des 0 et des 1, même si c# s'en charge pour vous, il est nécessaire de savoir comme il procède pour comprendre ce qu'il se passe en cas de problème. On retrouve donc dans la mémoire la représentation binaire des nombres entiers. Ainsi la plage de valeur d'un **byte**, encodée en binaire, est :

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 1111\ 1110, 1111\ 1111\}$$

Les nombres entiers positifs sont ceux qui commencent par un 0, ils sont représentés sur l'intervalle :

$$\{0000\ 0000, 0000\ 0001, 0000\ 0010, 0000\ 0011, \ldots, 0111\ 1100, 0111\ 1101, 0111\ 1110, 0111\ 1111\}$$

Les valeurs entières correspondantes sont :

$$\{0, 1, 2, \ldots, 125, 126, 127\}$$

Et les nombres négatifs, commençant par un 1, sont donc représentés sur l'intervalle :

$$\{1000\ 0000, 1000\ 0001, 1000\ 0010, 1000\ 0011, \ldots, 1111\ 1100, 1111\ 1101, 1111\ 1110, 1111\ 1111\}$$

Les nombres négatifs sont disposés du plus éloigné de 0 jusqu'au plus proche de 0, l'intervalle précédent code les valeurs :

$$\{-2^7, -(2^7-1), -(2^7-2), -(2^7-3), \ldots, -4, -3, -2, -1\}$$

Par conséquent, on représente avec un **byte** les valeurs

$$\{-2^7, -(2^7-1), -(2^7-2), -(2^7-3), \ldots, -4, -3, -2, -1, 0, 1, 2, \ldots, 125, 126, 127\}$$

Les opérations arithmétiques sont exécutées assez bêtement, si vous calculez 0111 1111 + 0000 0001, ce qui correspond à $(2^7 - 1) + 1$, le résultat mathématique est 2^7, ce qui se code 1000 0000, ce qui est le codage de -2^7. Soyez donc attentifs, en cas de dépassement de capacité d'un nombre entier, vous vous retrouverez avec des nombres qui ne veulent rien dire. Si vous souhaitez faire des calculs sur des réels, un type flottant sera davantage adapté.

Le principe de représentation des entiers est le même sur tous les types entiers. On résume cela dans le tableau suivant :

nom	taille (t)	nombre de valeurs (2^{8t})	plus petite valeur	plus grande valeur
byte	1 octet	2^8 valeurs	-2^7	$2^7 - 1$
short	2 octet	2^{16} valeurs	-2^{15}	$2^{15} - 1$
int	4 octets	2^{32} valeurs	-2^{31}	$2^{31} - 1$
long	8 octets	2^{64} valeurs	-2^{63}	$2^{63} - 1$

1.2.6 Nombre décimaux à point-fixe

Le type **decimal** permet de représenter des nombre dont le nombre de chiffres représentatifs après la virgule est fixé. Du fait de sa précision (8 octets), ce type est particulièrement apprécié pour représenter des quantités monétaires. 4 octets sont utilisés pour la partie entière et 4 pour la partie décimale. A votre avis, quelle est la plage de valeurs qu'il est possible de représenter avec ?

1.2.7 Flottants

Les flottants servent à représenter les réels. Leur nom vient du fait qu'on les représente de façon scientifique : un nombre décimal (à virgule) muni d'un exposant (un décalage de la virgule). Deux types de base servent à représenter les flottants :

nom	taille
float	4 octet
double	8 octets

Notez bien le fait qu'un point flottant étend le type à point fixe en permettant de "déplacer la virgule". Cela permet de représenter des valeurs très grandes ou très petites, mais au détriment de la précision. Nous examinerons dans les exercices les avantages et inconvénients des flottants.

Un littéral flottant s'écrit avec un point, par exemple l'approximation à 10^{-2} près de π s'écrit 3.14. Il est aussi possible d'utiliser la notation scientifique, par exemple le décimal 1 000 s'écrit `1e3`, à savoir 1.10^3. Nous nous limiterons à la description du type **float**, le type **double** étant soumis à des règles similaires. Attention, les littéraux de type **float** s'écrivent avec un f en suffixe.

Représentation en mémoire d'un float

Le codage d'un nombre de type **float** (32 bits) est découpé en trois parties :

partie	taille
le bit de signe	1 bit
l'exposant	8 bits
la mantisse	23 bits

Le nombre est positif si le bit de signe est à 0, négatif si le bit de signe est à 1. La mantisse et l'exposant sont codés en binaire, la valeur absolue d'un flottant de mantisse m et d'exposant e est $\frac{m}{2^{23}}.2^e$. Le plus grand entier qu'il est possible de coder sur 23 octets est $(2^{23}-1)$, comme les bits de la mantisse représentent la partie décimale du nombre que l'on souhaite représenter, on obtient le plus grand nombre pouvant être représenté par la mantisse en divisant $(2^{23}-1)$ par 2^{23}, soit $\frac{(2^{23}-1)}{2^{23}}$. Comme le plus grand exposant est 2^7 , le plus grand flottant est $\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$, donc le plus petit est $-\frac{(2^{23}-1)}{2^{23}}.2^{(2^7)}$.

1.2.8 Caractères

Un **char** sert à représenter le code `UNICODE` d'un caractère, il est donc codé sur 2 octets. Il est possible d'affecter à une telle variable toute valeur du code `UNICODE` entourée de simples quotes. Par exemple, l'affectation suivante place dans `a` le code `UNICODE` du caractère `'B'`.

```
char a;
a = 'B';
```

Si une variable numérique entière `e` contient une valeur `UNICODE`, on obtient le caractère correspondant avec (**char**)`e`. Inversement, on obtient la valeur `UNICODE` d'une variable de type caractère `c` avec l'expression (**int**)`c` (où **int** peut être remplacé par n'importe quel type numérique entier d'au moins 2 octets).

1.2.9 Chaînes de caractères

Une chaîne de caractères, se déclarant avec le mot-clé **string**, est une succession de caractères (aucun, un ou plusieurs). Les littéraux de ce type se délimitent par des double quotes, et l'instruction de saisie s'écrit sans le <type>.Parse. Par exemple,

```
string s = "toto";
string k = Console.ReadLine();
```

1.3 Opérateurs

1.3.1 Généralités

Opérandes et arité

Lorsque vous effectuez une opération, par exemple 3 + 4, le + est un opérateur, 3 et 4 sont des opérandes. Si l'opérateur s'applique à 2 opérandes, on dit qu'il s'agit d'un opérateur **binaire**, ou bien d'**arité** 2. Un opérateur d'arité 1, dit aussi **unaire**, s'applique à un seul opérande, par exemple `-x`, le `x` est l'opérande et le − unaire est l'opérateur qui, appliqué à `x`, nous donne l'opposé de celui-ci, c'est-à-dire le nombre qu'il faut additionner à `x` pour obtenir 0. Il ne faut pas le confondre avec le − binaire, qui appliqué à x et y, additionne à x l'opposé de y.

En C#, les opérateurs sont unaires ou binaires, et il existe un seul opérateur ternaire.

Associativité

Si vous écrivez une expression de la forme `a + b + c`, où `a`, `b` et `c` sont des variables entières, vous appliquez deux fois l'opérateur binaire + pour calculer la somme de 3 nombres `a`, `b` et `c`. Dans quel ordre ces opérations sont-elles effectuées ? Est-ce que l'on a `(a + b) + c` ou `a + (b + c)` ? Cela importe peu, car le + **entier** est **associatif**, ce qui signifie qu'il est possible de modifier le parenthèsage d'une somme d'entiers sans en changer le résultat. Attention : l'associativité est une rareté ! Peu d'opérateurs sont associatifs, une bonne connaissance des règles sur les priorités et le parenthèsage par défaut est donc requise.

Formes préfixes, postfixes, infixes

Un opérateur unaire est **préfixe** s'il se place avant son opérande, **postfixe** s'il se place après. Un opérateur binaire est **infixe** s'il se place entre ses deux opérandes (`a + b`), **préfixe** s'il se place avant (`+ a b`), **postfixe** s'il se place après (`a b +`). Vous rencontrez en `C` des opérateurs unaires préfixes et d'autres postfixes (on imagine difficilement un opérateur unaire infixe), par contre tous les opérateurs binaires seront infixe.

Priorités

Les règles des priorités en C# sont nombreuses et complexes, nous ne ferons ici que les esquisser. Nous appelerons **parenthèsage implicite** le parenthèsage adopté par défaut par le C#, c'est à dire l'ordre dans lequel il effectue les opérations. La première règle à retenir est qu'un opérateur unaire est **toujours prioritaire** sur un opérateur binaire.

1.3.2 Les opérateurs unaires

Négation arithmétique

La négation arithmétique est l'opérateur − qui à une opérande x, associe l'opposé de x, c'est-à-dire le nombre qu'il faut additionner à x pour obtenir 0.

Négation binaire

La négation binaire `~` agit directement sur les bits de son opérande, tous les bits à 0 deviennent 1, et vice-versa. Par exemple, `~`127 (tous les bits à 1 sauf le premier) est égal à 128 (le premier bit à 1 et tous les autres à 0).

Priorités

Tous les opérateurs unaires sont de priorité équivalente, le parenthèsage implicite est fait le plus à droite possible, on dit que ces opérateurs sont **associatifs à droite**. Par exemple, le parenthèsage implicite de l'expression `~-~i` est `~(-(~i))`. C'est plutôt logique : si vous parvenez à placer les parenthèses différement, prevenez-moi parce que je ne vois pas comment faire...

1.3.3 Les opérateurs binaires

Opérations de décalages de bits

L'opération `a` >> 1 effectue un décalage des bits de la représentation binaire de `a` vers la droite. Tous les bits sont décalés d'un cran vers la droite, le dernier bit disparaît, le premier prend la valeur 0. L'opération `a` << 1 effectue un décalage des bits de la représentation binaire de `a` vers la gauche. Tous les bits sont décalés d'un cran vers la gauche, le premier bit disparaît et le dernier devient 0. Par exemple, 32 << 2 associe à 0010 0000 << 2 la valeur dont la représentation binaire 1000 0000 et 32 >> 3 associe à 0010 0000 >> 3 la valeur dont la représentation binaire 0000 0100.

Opérations logiques sur la représentation binaire

L'opérateur & associe à deux opérandes le ET logique de leurs représentations binaires par exemple 60 & 15 donne 12, autrement formulé 0011 1100 `ET` 0000 1111 = 0000 1100. L'opérateur | associe à deux opérandes le OU logique de leurs représentations binaires par exemple 60 | 15 donne 63, autrement formulé 0011 1100 `OU` 0000 1111 = 0011 1111. L'opérateur ^ associe à deux opérandes le OU **exclusif** logique de leurs représentations binaires par exemple 60 ^ 15 donne 51, autrement formulé 0011 1100 `OU EXCLUSIF` 0000 1
Deux ^ successifs s'annulent, en d'autres termes `a^b^b=a`.

Affectation

Ne vous en déplaise, le = est bien un opérateur binaire. Celui-ci affecte à l'opérande de gauche (appelée `Lvalue` par le compilateur), qui doit être une variable, une valeur calculée à l'aide d'une expression, qui est l'opérande de droite. Attention, il est possible d'effectuer une affectation pendant l'évaluation d'une expression. Par exemple,

```
a = b + (c = 3);
```

Cette expression affecte à `c` la valeur 3, puis affecte à `a` la valeur `b` + `c`.

Priorités

Tous les opérateurs binaires ne sont pas de priorités équivalentes. Ceux de priorité la plus forte sont les opérateurs arithmétiques (∗, /, %, +, −), puis les opérateus de décalage de bit (<<, >>), les opérateurs de bit (&, ^, |), et enfin l'affectation =. Représentons dans une tableau les opérateurs en fonction de leur priorité, plaçons les plus prioritaire en haut et les moins prioritaires en bas. Parmi les opérateurs arithmétiques, les multiplications et divisions sont prioritaires sur les sommes et différence :

noms	opérateurs
produit	∗, /, %
sommes	+, −

Les deux opérateurs de décalage sont de priorité équivalente :

noms	opérateurs
décalage binaire	>>, <<

L'opérateur & est assimilé à un produit, | à une somme. Donc , & est prioritaire sur |. Comme ^ se trouve entre les deux, on a

noms	opérateurs
ET binaire	&
OU Exlusif binaire	^
OU binaire	|

Il ne nous reste plus qu'à assembler les tableaux :

noms	opérateurs
produit	$*$, /, %
somme	+, −
décalage binaire	>>, <<
ET binaire	&
OU Exlusif binaire	^
OU binaire	|
affectation	=

Quand deux opérateurs sont de même priorité le parenthèsage implicite est fait le plus à **gauche possible**, on dit que ces opérateurs sont **associatifs à gauche**. Par exemple, le parenthèsage implicite de l'expression `a - b - c` est

```
(a - b) - c
```

et **certainement pas** `a - (b - c)`. Ayez donc cela en tête lorsque vous manipulez des opérateurs non associatifs !

Attention : la seule exception est le =, qui est associatif à droite. Par exemple,

```
a = b = c;
```

se décompose en `b = c` suivi de `a = b`.

1.3.4 Formes contractées

Le C# étant un dérivé du C, qui est un langage de paresseux, tout à été fait pour que les programmeurs aient le moins de caractères possible à saisir. Je vous préviens : j'ai placé ce chapitre pour que soyez capable de décrypter la bouillie que pondent certains programmeurs, pas pour que vous les imitiez ! Alors vous allez me faire le plaisir de faire usage des formes contractées avec parcimonie, n'oubliez pas qu'il est très important que votre code soit **lisible**.

Unaires

Il est possible d'**incrémenter** (augmenter de 1) la valeur d'une variable `i` en écrivant `i++`, ou bien `++i`. De la même façon on peut **décrémenter** (diminuer de 1) `i` en écrivant `i--` (forme postfixe), ou bien `--i` (forme préfixe). Vous pouvez décider d'incrémenter (ou de décrémenter) la valeur d'une variable pendant un calcul, par exemple,

```
a = 1;
b = (a++) + a;
```

évalue successivement les deux opérandes `a++` et `a`, puis affecte leur somme à `b`. L'opérande `a++` est évaluée à 1, puis est incrémentée, donc lorsque la deuxième opérande `a` est évaluée, sa valeur est 2. Donc la valeur de b après l'incrémentation est 3. L'incrémentation contractée sous forme postfixe s'appelle une **post-incrémentation**. Si l'on écrit,

```
a = 1;
b = (++a) + a;
```

On opère une **pré-incrémentation**, `++a` donne lieu à une incrémentation avant l'évaluation de `a`, donc la valeur 4 est affectée à `b`. On peut de façon analogue effectuer une **pré-decrémentation** ou a **post-decrémentation**. Soyez très attentifs au fait que ce code n'est pas portable, il existe des compilateurs qui évaluent les opérandes dans le désordre ou diffèrent les incrémentations, donnant ainsi des résultats autres que les résultats théoriques exposés précédement. Vous n'utiliserez donc les **incrémentation** et **decrémentation** contractées que lorsque vous serez certain que l'ordre d'évaluation des opérandes ne pourra pas influer sur le résultat.

Binaires

Toutes les affectations de la forme `variable = variable operateurBinaire expression` peuvent être contractées sous la forme `variable operateurBinaire= expression`. Par exemple,

avant	après
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a >> i`	`a >>= i`
`a = a << i`	`a <<= i`
`a = a & b`	`a &= b`
`a = a ^ b`	`a ^= b`
`a = a | b`	`a |= b`

Vous vous douterez que l'égalité ne peut pas être contractée...

1.3.5 Opérations hétérogènes

Le fonctionnement par défaut

Nous ordonnons de façon grossière les types de la façon suivante : **double** > **float** > **long** > **int** > **short** > **byte**. Dans un calcul où les opérandes sont de types hétérogènes, l'opérande dont le type T est de niveau le plus élévé (conformément à l'ordre énoncé ci-avant) est sélectionné et l'autre est converti dans le type T.

Le type **decimal** est à part dans le sens où aucune conversion implicite n'est prévu. Les chapitres suivants vous donneront des clés pour effectuer des conversions quand même.

Le problème

Il se peut cependant que dans un calcul, cela ne convienne pas. Si par exemple, vous souhaitez calculer l'inverse $\frac{1}{x}$ d'un nombre entier x, et que vous codez

```
int i = 4;
Console.WriteLine("L'inverse de " + i + " est " + 1/i);
```

Vous constaterez que résultat est inintéressant au possible. En effet, comme i et 1 sont tout deux de type entier, c'est la division entière est effectuée, et de toute évidence le résultat est 0. Ici la bidouille est simple, il suffit d'écrire le 1 avec un point :

```
int i = 4;
Console.WriteLine("L'inverse de " + i + " est " + 1./i);
```

Le compilateur, voyant un opérande de type flottant, convertit lors du calcul l'autre opérande, `i`, en flottant. De ce fait, c'est une division flottante et non entière qui est effectuée. Allons plus loin : comment faire pour appliquer une division flottante à deux entiers ? Par exemple :

```
int i = 4, j= 5;
Console.WriteLine("Le quotient de " + i + " et " + j + " est " + i/j + ".");
```

Cette fois-ci c'est inextricable, vous pouvez placer des points où vous voudrez, vous n'arriverez pas à vous débarrasser du warning et ce programme persistera à vous dire que ce quotient est -0.000000 ! Une solution particulièrement crade serait de recopier `i` et `j` dans des variables flottantes avant de faire la division, une autre méthode de bourrin est de calculer `(i + 0.)/j`. Mais j'espère que vous réalisez que seuls les boeufs procèdent de la sorte.

Le `cast`

Le seul moyen de vous sortir de là est d'effectuer un `cast`, c'est à dire une conversion de type sur commande. On caste en plaçant entre parenthèse le type dans lequel on veut convertir juste avant l'opérande que l'on veut convertir. Par exemple,

```
int i = 4, j= 5;
Console.WriteLine("Le quotient de " + i + " et " + j +
        " est " + (float)i/j + ".");
```

Et là, ça fonctionne. La variable valeur contenue dans `i` est convertie en **float** et de ce fait, l'autre opérande, `j`, est aussi convertie en **float**. La division est donc une division flottante. Notez bien que le `cast` est un opérateur unaire, donc prioritaire sur la division qui est un opérateur binaire, c'est pour ça que la conversion de `i` a lieu avant la division. Mais si jamais il vous vient l'idée saugrenue d'écrire

```
int i = 4, j= 5;
Console.WriteLine("Le quotient de " + i + " et " + j +
        " est " + (float)(i/j) + ".");
```

Vous constaterez très rapidement que c'est une alternative peu intelligente. En effet, le résulat est flottant, mais comme la division a lieu avant toute conversion, c'est le résultat d'une division entière qui est converti en flottant, vous avez donc le même résultat que si vous n'aviez pas du tout casté.

1.3.6 Les priorités

Ajoutons le `cast` au tableau des priorités de nos opérateurs :

noms	opérateurs
opérateurs unaires	`cast`, $-$, $\tilde{}$, $++$, $--$
produit	$*$, $/$, $\%$
somme	$+$, $-$
décalage binaire	$>>$, $<<$
ET binaire	$\&$
OU Exlusif binaire	$\hat{}$
OU binaire	$\mid$
affectation	$=$

1.4 Traitements conditionnels

On appelle traitement conditionnel un portion de code qui n'est pas exécutée systématiquement, c'est à dire des instructions dont l'exécution est conditionnée par le succès d'un test.

1.4.1 Si ... Alors

Principe

En algorithmique un traitement conditionnel se rédige de la sorte :

Si *condition* **alors**
| instructions
fin si

Si la condition est vérifiée, alors les instructions sont exécutées, sinon, elles ne sont pas exécutées. L'exécution de l'algorithme se poursuit alors en ignorant les instructions se trouvant entre le `alors` et le `finSi`. Un traitement conditionnel se code de la sorte :

```
if (<condition>)
{
        <instructions>
}
```

Notez bien qu'il n'y a pas de point-virgule après la parenthèse du **if**.

Comparaisons

La formulation d'une condition se fait souvent à l'aide des opérateurs de comparaison. Les opérateurs de comparaison disponibles sont :

- == : égalité
- != : non-égalité
- <, <= : inférieur à, respectivement strict et large
- >, >= : supérieur à, respectivement strict et large

Par exemple, la condition `a == b` est vérifiée si et seulement si `a` et `b` ont la même valeur au moment où le test est évalué. Par exemple,

```
using System;

namespace tests
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        Console.WriteLine("Saisissez une valeur");
                        int i = int.Parse(Console.ReadLine());
                        if (i == 0)
                        {
                                Console.WriteLine("Vous avez saisi une valeur nulle.")
                        }
                        Console.WriteLine("Au revoir !");
                }
        }
}
```

Si au moment où le test `i == 0` est évalué, la valeur de `i` est bien 0, alors le test sera vérifié et l'instruction `Console.WriteLine("Vous avez saisi une valeur nulle.");` sera bien exécutée. Si le test n'est pas vérifié, les instructions du bloc sous la portée du **if** sont ignorées.

Si ... Alors ... Sinon

Il existe une forme étendue de traitement conditionnel, on la note en algorithmique de la façon suivante :

Si *condition* **alors**
| instructions
sinon
| autresinstructions
fin si

Les instructions délimitées par `alors` et `sinon` sont exécutées si le test est vérifié, et les instructions délimitées par `sinon` et `finSi` sont exécutées si le test n'est pas vérifié. On traduit le traitement conditionnel étendu de la sorte :

```
if (<condition>)
{
        <instructions1>
}
else
{
        <instructions2>
}
```

Par exemple,

```
using System;

namespace tests
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        Console.WriteLine("Saisissez une valeur");
                        int i = int.Parse(Console.ReadLine());
                        if (i == 0)
                        {
                                Console.WriteLine("Vous avez saisi une valeur nulle.")
                        }
                        else
                        {
                                Console.WriteLine("La valeur que vous avez saisi, " +
                                                  ", n'est pas nulle.");
                        }
                }
        }
}
```

Notez la présence de l'opérateur de comparaison ==. **Si vous utilisez = pour comparer deux valeurs, ça ne compilera pas !**

Connecteurs logiques

On formule des conditions davantage élaborées en utilisant des connecteurs `et` et `ou`. La condition `A et B` est vérifiée si les deux conditions `A` et `B` sont vérifiées simultanément. La condition `A ou B` est vérifiée si au moins une des deux conditions `A` et `B` est vérifiée. Le `et` s'écrit && et le `ou` s'écrit ||. Par exemple, voici un programme C qui nous donne le signe de $i \times j$ sans les multiplier.

```
public static void Main (string[] args)
{
        Console.WriteLine("Saisissez deux valeurs numériques : ");
        float i = float.Parse(Console.ReadLine());
        float j = float.Parse(Console.ReadLine());
        Console.Write("Le produit de " + i + " par " + j + " est ");
        if ((i >= 0 && j >= 0) || (i < 0 && j < 0))
        {
                Console.WriteLine("positif.");
        }
        else
        {
                Console.WriteLine("négatif.");
        }
}
```

Accolades superflues

Lorsqu'une seule instruction d'un bloc **if** doit être exécutée, les accolades ne sont plus nécessaires. Il est possible par exemple de reformuler le programme précédent de la sorte :

```
public static void Main (string[] args)
{
        Console.WriteLine("Saisissez deux valeurs numériques : ");
        float i = float.Parse(Console.ReadLine());
        float j = float.Parse(Console.ReadLine());
        Console.Write("Le produit de " + i + " par " + j + " est ");
        if ((i >= 0 && j >= 0) || (i < 0 && j < 0))
                Console.WriteLine("positif.");
        else
                Console.WriteLine("négatif.");
}
```

Blocs

Un **bloc** est un ensemble d'instructions délimité par des accolades. Par exemple le bloc **public static void** `Main(str` ou encore le bloc **if**. Vous aurez remarqué qu'il est possible d'imbriquer les blocs et qu'il convient d'ajouter un niveau d'indentation supplémentaire à chaque fois qu'un nouveau bloc est ouvert.

Une des conséquences de la structure de blocs du langage c# s'observe dans l'exemple suivant :

```
public static void Main (string[] args)
{
        int i =  4;
        if (i == 4)
        {
                int j = i + 1;
        }
        Console.WriteLine("j = " + j + ".");
}
```

Vous aurez remarqué la variable j est déclarée à l'intérieur du bloc **if**, et utilisé à l'extérieur de ce bloc. Ce code ne compilera pas parce que **une variable n'est utilisable qu'à l'intérieur du bloc où elle a été déclarée**. La **portée** (ou encore la **visibilité**) de la variable j se limite donc à ce bloc **if**.

Opérateur ternaire

En plaçant l'instruction suivante à droite d'une affectation,

```
<variable> = (<condition>) ? <valeur> : <autrevaleur> ;
```

on place `valeur` dans `variable` si `condition` est vérifié, `autrevaleur` sinon. Par exemple,

```
max = (i>j) ? i : j ;
```

place la plus grande des deux valeurs `i` et `j` dans `max`. Plus généralement on peut utiliser le si ternaire dans n'importe quel calcul, par exemple

```
public static void Main (string[] args)
{
        int i = 4;
        int j = 2;
        int k = 7;
        int l;
        Console.WriteLine((i> (l = (j>k) ? j : k)) ? i : l);
}
```

`j = (j>k) ? j : k)` place dans `l` la plus grande des valeurs `j` et `k`, donc `(i > (l = (j > k) ? j : k)) ? i : l` est la plus grande des valeurs `i`, `j` et `k`. La plus grande de ces trois valeurs est donc affichée par cette instruction.

1.4.2 Switch

Le **switch** est une instruction permettant sélectionner un cas selon la valeur d'une variable. La syntaxe est la suivante :

```
switch(<nomvariable>)
{
  case <valeur_1> : <instructions_1> ; break ;
  case <valeur_2> : <instructions_2> ; break ;
  /* ... */
  case <valeur_n> : <instructions_n> ; break ;
  default : <instructionspardefaut> ; break;
}
```

Si *nomvariable* contient la valeur $valeur_i$, alors les $instructions_i$ sont exécutées. Si aucune des valeurs énumérées ne correspond à celle de *nomvariable*, ce sont les *instructionspardefaut* qui sont exécutées. Les **break** servent à fermer chaque cas, y compris le dernier ! Si par exemple, nous voulons afficher le nom d'un mois en fonction de son numéro, on écrit :

```
switch(numeroMois)
{
  case 1 : Console.Write("janvier") ; break ;
  case 2 : Console.Write("fevrier") ; break ;
  case 3 : Console.Write("mars") ; break ;
  case 4 : Console.Write("avril") ; break ;
  case 5 : Console.Write("mai") ; break ;
  case 6 : Console.Write("juin") ; break ;
  case 7 : Console.Write("juillet") ; break ;
```

```
  case 8 : Console.Write("aout") ; break ;
  case 9 : Console.Write("septembre") ; break ;
  case 10 : Console.Write("octobre") ; break ;
  case 11 : Console.Write("novembre") ; break ;
  case 12 : Console.Write("decembre") ; break ;
  default : Console.Write("Je connais pas ce mois...");break;
}
```

1.4.3 Booléens

Une variable booléenne ne peut prendre que deux valeurs : *vrai* et *faux*. Le type booléen en c# est **bool** et une variable de ce type peut prendre soit la valeur **true**, soit la valeur **false**.

Utilisation dans des if

Lorsqu'une condition est évaluée, par exemple lors d'un test, cette condition prend à ce moment la valeur *vrai* si le test est vérifié, *faux* dans le cas contraire. Il est donc possible de placer une variable booléenne dans un if. Observons le test suivant :

```
public static void Main (string[] args)
{
        Console.WriteLine("Saisissez un booléen : ");
        bool b = bool.Parse(Console.ReadLine());
        if (b)
                Console.WriteLine("b is true.");
        else
                Console.WriteLine("b is false.");
}
```

Si *b* contient la valeur **true**, alors le test est réussi, sinon le **else** est exécuté. On retiendra donc qu'il est possible de placer dans un **if** toute expression pouvant prendre les valeurs **true** ou **false**.

Tests et affectations

Un test peut être effectué en dehors d'un **if**, par exemple de la façon suivante :

```
bool x = (3>2);
```

On remarque que (3>2) est une condition. Pour décider quelle valeur doit être affectée à `x`, cette condition est évaluée. Comme dans l'exemple ci-dessus la condition est vérifiée, alors elle prend la valeur **true**, et cette valeur est affectée à `x`.

Connecteurs logiques binaires

Les connecteurs || et && peuvent s'appliquer à des valeurs (ou variables) booléennes. Observons l'exemple suivant :

```
bool x = (true && false) || (true);
```

Il s'agit de l'affectation à `x` de l'évaluation de la condition (**true** && **false**) || (**true**). Comme (**true** && **false**) a pour valeur **false**, la condition **false** || **true** est ensuite évaluée et prend la valeur **true**. Donc la valeur **true** est affectée à `x`.

Opérateur de négation

Parmi les connecteurs logiques se trouve !, dit opérateur de négation. La négation d'une expression est vraie si l'expression est fausse, fausse si l'expression est vraie. Par exemple,

```
bool x = !(3==2);
```

Comme 3 == 2 est faux, alors sa négation !(3 == 2) est vraie. Donc la valeur **true** est affectée à `x`.

1.4.4 Les priorités

Complétons notre tableau des priorités en y adjoignant les connecteurs logiques et les opérateurs de comparaison :

noms	opérateurs
opérateurs unaires	`cast`, −, ˜, !, ++, −−
produit	∗, /, %
somme	+, −
décalage binaire	>>, <<
comparaison	>, <, >=, <=
égalité	==, !=
ET binaire	&
OU Exlusif binaire	ˆ
OU binaire	|
connecteurs logiques	&&, ||
if ternaire	()?:
affectations	=, +=, −=, ...

1.5 Boucles

Nous souhaitons créer un programme qui nous affiche tous les nombres de 1 à 10, donc dont l'exécution serait la suivante :

```
1 2 3 4 5 6 7 8 9 10
```

Une façon particulièrement vilaine de procéder serait d'ecrire 10 `Console.WriteLine` successifs, avec la laideur des copier/coller que cela impliquerait. Nous allons étudier un moyen de coder ce type de programme avec un peu plus d'élégance.

1.5.1 Définitions et terminologie

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cette ensemble d'instructions s'appelle le **corps** de la boucle. Chaque exécution du corps d'une boucle s'appelle une **itération**, ou plus informellement un **passage** dans la boucle. Lorsque l'on s'apprête à exécuter la première itération, on dit que l'on **rentre** dans la boucle, lorsque la dernière itération est terminée, on dit qu'on **sort** de la boucle. Il existe trois types de boucle :

- **while**
- **do** ... **while**
- **for**

Chacune de ces boucles a ses avantages et ses inconvénients. Nous les passerons en revue ultérieurement.

1.5.2 while

En C#, la boucle `tant que` se code de la façon suivante :

```
while(<condition>)
{
  <instructions>
}
```

Les instructions du corps de la boucle sont délimitées par des accolades. La condition est évaluée **avant** chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Un fois que la condition n'est plus vérifiée, l'exécution se poursuit après l'accolade fermante. Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant,

```
public static void Main (string[] args)
{
        int i = 1;
        while (i <= 5)
        {
                Console.Write(i + " ");
                i++;
        }
        Console.Write("\n");
}
```

Ce programme **initialise** i **à** 1 et tant que la valeur de i n'éxcède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante** ! Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre programme ne se comportera pas du tout comme prévu. Notez bien par ailleurs qu'il n'y a **pas de point-virgule après le while !**

1.5.3 do ... while

Voici la syntaxe de cette boucle :

```
do
{
  <instructions>
}
while(<condition>);
```

La fonctionnement est analogue à celui de la boucle `tant que` à quelques détails près :
- la condition est évaluée **après** chaque passage dans la boucle.
- On exécute le corps de la boucle **tant que** la condition est vérifiée.

En C, la boucle `répéter ... jusqu'à` est en fait une boucle `répéter ... tant que`, c'est-à-dire une boucle `tant que` dans laquelle la condition est évaluée **à la fin**. Une boucle **do** ... **while** est donc exécutée donc **au moins une fois**. Reprennons l'exemple précédent avec une boucle **do** ... **while** :

```
public static void Main (string[] args)
{
        int i = 1;
        do
        {
                Console.Write(i + " ");
                i++;
        }
        while (i <= 5);
        Console.Write("\n");
}
```

De la même façon que pour la boucle **while**, le compteur est initialisé avant le premier passage dans la boucle. Un des usages les plus courant de la boucle **do** ... **while** est le contrôle de saisie :

```
public static void Main (string[] args)
{
        int i;
        do
        {
                Console.Write("Saisissez un entier positif ou nul : ");
                i = int.Parse(Console.ReadLine());
                if (i < 0)
                        Console.WriteLine("J'ai dit positif ou nul !");
        }
        while (i < 0);
        Console.WriteLine("Vous avez saisi " + i + ".");
}
```

1.5.4 for

Cette boucle est quelque peu délicate. Commençons par donner sa syntaxe :

```
for(<initialisation> ; <condition> ; <pas>)
{
  <instructions>
}
```

L'`<initialisation>` est une instruction exécutée avant le premier passage dans la boucle. La `<condition>` est évaluée **avant** chaque passage dans la boucle, si elle n'est pas vérifiée, on ne passe pas dans la boucle

et l'exécution de la boucle pour est terminée. La `<pas>` est une instruction exécutée **après** chaque passage dans la boucle. On peut convertir une boucle **for** en boucle **while** en procédant de la sorte :

```
<initialisation>
while(<condition>)
{
  <instructions>
  <pas>
}
```

On re-écrit l'affiche des 5 premiers entiers de la sorte en utilisant le fait que `<initialisation>` = `i` = 1, `<condition>` = `i` <= 5 et `<pas>` = `i++`. On obtient :

```
public static void Main (string[] args)
{
        for (int i = 1; i <= 5 ;i++)
        {
                Console.Write(i + " ");
        }
        Console.Write("\n");
}
```

On utilise une boucle **for** lorsque l'on connait en entrant dans la boucle combien d'itérations devront être faites. Par exemple, n'utilisez pas une boucle `pour` pour contrôler une saisie !

1.5.5 Accolades superflues

De la même façon qu'il est possibl de supprimer des accolades autour d'une instruction d'un bloc **if**, on peut supprimer les accolades autour du corps d'une boucle si elle ne contient qu'une seule instruction.

1.6 Chaînes de caractères

1.6.1 Exemple

Etant donné le programme dont l'exécution est tracée ci dessous :

```
Saisissez une phrase :
Les framboises sont perchees sur le tabouret de mon grand-pere.
Vous avez saisi :
Les framboises sont perchees sur le tabouret de mon grand-pere.
Cette phrase commence par une majuscule.
Cette phrase se termine par un point.
```

Pourriez-vous mettre au point un tel programme ?

1.6.2 Définition

Une **chaîne de caractères** est une suite de `char` (caractères UNICODE). Le type utilisé pour déclarer une chaîne de caractères est **string**.

1.6.3 Déclaration et initialisation

Une chaîne se déclare de la sorte :

```
string <nom_chaine> = "valeur initiale";
```

Par exemple, on déclare une chaine `c` contenant "nabucodonosor" de la sorte :

```
string c = "nabuchodonosor";
```

1.6.4 Opérations

Concaténation

La concaténation de deux chaînes est la juxtaposition des caractères qui les composent. L'opérateur utilisé pour ce faire est +.

Par exemple :

```
string a = "nabucho";
string b = "donosor";
string c = a + b;
```

Longueur

La longueur d'une chaîne `s` (i.e. le nombre de caractères qu'elle contient d'obtient avec l'attribut `.Length`. Par exemple,

```
Console.WriteLine("La longueur de " + s + " est " + s.Length + ".");
```

Extraction de caractères

On extrait le i-ème caractère d'une chaîne `s` avec l'instruction `s[i]` (attention les caractères sont indicés à partir de 0). Par exemple,

```
string s = "nabuchodonosor";
char c = s[1];
Console.WriteLine("Le deuxième caractère de " + s + " est " + c + ".");
```

Nous sommes maintenant en mesure de coder l'exemple donné au début du cours :

```
using System;

namespace exempleCours
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        Console.WriteLine("Saisissez une phrase :");
                        string s = Console.ReadLine();
                        int n = s.Length;
                        Console.Write("Cette phrase ");
                        if (s[0] >= 'A' && s[0] <= 'Z')
                                Console.Write("commence");
                        else
                                Console.Write("ne commence pas");
                        Console.WriteLine(" par une majuscule.");
                        Console.Write("Cette phrase ");
                        if (s[n - 1] == '.')
                                Console.Write("se termine");
                        else
                                Console.Write("ne se termine pas");
                        Console.WriteLine(" par un point.");
                }
        }
}
```

Extraction de sous-chaînes

On extrait de s une sous-chaîne de j caractères à partir du caractère d'indice i en utilisant `s.Substring(i, j)`. Par exemple,

```
string s = "nabuchodonosor";
string s2 = s.Substring(4, 7);
```

`s2` contient alors *chodono*.

1.6.5 Chaînes modifiables

Le problème se posant avec les chaînes en c# est qu'elle ne sont pas modifiables. Le type `StringBuilder` permet de créer des chaînes modifiables.

Pour utilisez des `StringBuilder`, vous devez ajouter **using** `System.Text`; au début de votre code source.

Déclaration

La syntaxe permettant de créer une chaîne vide est

```
StringBuilder s = new StringBuilder(n);
```

où `n` est le nombre maximal de caractères que pourra contenir la chaîne.

Vous avez aussi la possibilité de recopier une **string** dans une `StringBuilder` en procédant de la sorte :

```
StringBuilder sb = new StringBuilder(s);
```

Insertion

Vous pouvez insérer un caractère dans une `StringBuilder` s en utilisant l'instruction `s.Insert(i, a)`. Les caractères se trouvant après l'indice i dans s sont décalés d'une position vers la droite et a est placé à l'indice i.

Modification

Vous pouvez remplacer le caractère d'indice i dans s par a avec l'instruction `s[i] = a`. Il faut bien entendu qu'il y ait déjà un caractère à cet indice dans s, sinon le programme plante...

Conversions

Pour effectuer des conversions entre **string** et `StringBuilder`, inutile d'effectuer des casts, ça ne fonctionnera pas.

- On convertit `s` de type **string** vers `StringBuilder` avec **new** `StringBuilder(s)`
- et on convertit `sb` de type `StringBuilder` vers **string** avec `sb.ToString()`.

1.7 Tableaux

Considérons un programme dont l'exécution donne :

```
Saisissez dix valeurs :
1 : 4
2 : 7
3 : 34
4 : 1
5 : 88
6 : 22
7 : 74
8 : 19
9 : 3
10 : 51
Saisissez une valeur :
22
22 est la 6-eme valeur saisie.
```

Comment programmer cela sans utiliser 10 variables pour stocker les dix premières valeurs saisies ?

1.7.1 Définitions

Une tableau est un regroupement de variables de même type, il est identifié par un nom. Chacune des variables du tableau est numérotée, ce numéro s'appelle un **indice**. Chaque variable du tableau est donc caractérisée par le nom du tableau et son indice.

Si par exemple, T est un tableau de 10 variables, alors chacune d'elles sera numérotée et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable. Les différentes variables de T porteront des numéros de 0 à 9, et nous appellerons chacune de ces variables un **élément** de T.

Une variable n'étant pas un tableau est appelée variable **scalaire**, un tableau par opposition à une variable scalaire est une variable **non scalaire**.

1.7.2 Déclaration

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la délaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient. La syntaxe est :

```
<type>[] <nomdutableau> = new <type>[<taille>];
```

Par exemple,

```
int[] T = new int[4];
```

déclare un tableau `T` contenant 4 variables de type **int**.

Attention ! `<type>[]` `<nomdutableau>` n'est pas un tableau ! C'est une variable contenant un tableau. Cela signifie que tant que vous ne lui avez pas affecté **new** `<type>[<taille>]`, vous ne pouvez pas l'utiliser.

1.7.3 Initialisation

Il est possible d'initialiser les éléments d'un tableau à la déclaration, on fait cela comme pour des variables scalaires :

```
<type> <nom> = <valeur d initialisation>;
```

La seule chose qui change est la façon d'écrire la valeur d'initialisation, on ecrit entre accolades tous les éléments du tableau, on les dispose par ordre d'indice croissant en les séparant par des virgules. La syntaxe générale de la valeur d'initialisation est donc :

```
<type>[] <nom> = new <type>[<taille>]{<valeur_0>, <valeur_1>, ..., <valeur_n-1>};
```

Par exemple, on crée un tableau contenant les 5 premiers nombres impairs de la sorte :

```
int[] T = new int[5]{1, 3, 5, 7, 9};
```

Lorsqu'un tableau est initialisé à la déclaration, il est même possible d'omettre sa taille :

```
int[] T = new int[]{1, 3, 5, 7, 9};
```

... voire une partie de sa déclaration :

```
int[] T = {1, 3, 5, 7, 9};
```

1.7.4 Accès aux éléments

Les éléments d'un tableau à n éléments sont indicés de 0 à $n-1$. On note `T[i]` l'élément d'indice i du tableau `T`. Les cinq éléments du tableau de l'exemple ci-avant sont donc notés `T[0]`, `T[1]`, `T[2]`, `T[3]` et `T[4]`.

1.7.5 Exemple

Nous pouvons maintenant mettre en place le programme du début du cours. Il est nécessaire de stocker 10 valeurs de type entier, nous allons donc déclarer un tableau `e` de la sorte :

```
int[] e = new int[10];
```

La déclaration ci-dessus est celle d'un tableau de 10 **int** appelé `e`. Il convient ensuite d'effectuer les saisies des 10 valeurs. On peut par exemple procéder de la sorte :

```
Console.Write("Saisissez dix valeurs : \n");
Console.Write("1 : ");
e[0] = int.Parse(Console.ReadLine());
Console.Write("2 : ");
e[1] = int.Parse(Console.ReadLine());
Console.Write("3 : ");
e[2] = int.Parse(Console.ReadLine());
Console.Write("4 : ");
e[3] = int.Parse(Console.ReadLine());
Console.Write("5 : ");
e[4] = int.Parse(Console.ReadLine());
Console.Write("6 : ");
e[5] = int.Parse(Console.ReadLine());
Console.Write("7 : ");
e[6] = int.Parse(Console.ReadLine());
Console.Write("8 : ");
e[7] = int.Parse(Console.ReadLine());
Console.Write("9 : ");
```

```
e[8] = int.Parse(Console.ReadLine());
Console.Write("10 : ");
e[9] = int.Parse(Console.ReadLine());
```

Les divers copier/coller necéssaires pour rédiger un tel code sont d'une laideur à proscrire. Nous procéderons plus élégament en faisant une boucle :

```
Console.WriteLine("Saisissez dix valeurs :");
for(int i = 0 ; i < 10 ; i++)
{
  Console.Write(i+1 + " : ");
  e[i] = int.Parse(Console.ReadLine());
}
```

Ce type de boucle s'appelle un **parcours** de tableau. En règle générale on utilise des boucles pour manier les tableaux, celles-ci permettent d'effectuer un traitement sur chaque élément d'un tableau. Ensuite, il faut saisir une valeur à rechercher dans le tableau :

```
Console.WriteLine("Saisissez une valeur :");
t = int.Parse(Console.ReadLine());
```

Nous allons maintenant rechercher la valeur `t` dans le tableau `e`. Considérons pour ce faire la boucle suivante :

```
i = 0;
while (e[i] != t)
        i++;
```

Cette boucle parcourt le tableau jusqu'à trouver un élément de `e` qui ait la même valeur que `t`. Le problème qui pourrait se poser est que si `t` ne se trouve pas dans le tableau `e`, alors la boucle pourrait ne pas s'arrêter. Si `i` prend des valeurs strictement plus grandes que 9, alors il se produira ce que l'on appelle un **débordement d'indice**. Vous devez toujours veiller à ce qu'il ne se produise pas de débordement d'indice ! Nous allons donc faire en sorte que la boucle s'arrête si `i` prend des valeurs strictement supérieures à 9.

```
i = 0;
while (i < 10 && e[i] != t)
        i++;
```

Il existe donc deux façons de sortir de la boucle :

- En cas de débordement d'indice, la condition `i` < 10 ne sera pas vérifiée. Une fois sorti de la boucle, `i` aura la valeur 10.
- Dans le cas où `t` se trouve dans le tableau à l'indice `i`, alors la condition `e[i] != t` ne sera pas vérifiée et on sortira de la boucle. Un fois sorti de la boucle, `i` aura comme valeur l'indice de l'élément de `e` qui est égal à `t`, donc une valeur comprise entre 0 et 9.

On identifie donc la façon dont on est sorti de la boucle en testant la valeur de `i` :

```
if (i == 10)
  Console.WriteLine(t + " ne fait pas partie des dix valeurs saisies.");
else
  Console.WriteLine(t + " est la " + (i + 1) + "-eme valeur saisie.");
```

Si (`i` == 10), alors nous sommes sorti de la boucle parce que l'élément saisi par l'utilisateur ne trouve pas dans le tableau. Dans le cas contraire, `t` est la `i`+1-ème valeur saisie par l'utilisateur. On additionne 1 à l'indice parce que l'utilisateur ne sait pas que dans le tableau les éléments sont indicés à partir de 0. Récapitulons :

```
using System;
```

```
namespace TPTableaux
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        int[] e = new int[10];
                        int i;
                        Console.WriteLine("Saisissez 10 valeurs : ");
                        for (i = 0 ; i < 10 ; i++)
                        {
                                Console.Write (i + 1 + " : ");
                                e[i] = int.Parse (Console.ReadLine());
                        }
                        Console.WriteLine("Saisissez une valeur : ");
                        int t = int.Parse (Console.ReadLine());
                        i = 0;
                        while(i < 10 && e[i] != t)
                                i++;
                        if (i == 10)
                                Console.WriteLine (t + " ne fait pas partie des dix v
                        else
                                Console.WriteLine (t + " est la " + (i + 1)+ "–Ã¨me v
                }
        }
}
```

1.7.6 L'instruction foreach

Il est possible de parcourir un tableau de façon séquentielle et en lecture seule grâce à l'instruction **foreach**.

```
foreach(<type> <variable> in <tableau>)
                <instructions>
```

Cette instruction affecte à <variable> pour chaque itération de la boucle un élément de <tableau>. On remarque au passage que le <type> doit être celui des éléments de <tableau>. Par exemple, le programme suivant affiche les quatre premières lettres de l'alphabet :

```
char[] t = {'a', 'b', 'c', 'd'};
foreach(char c in t)
                Console.WriteLine(c);
```

1.8 Sous-programmes

1.8.1 Les procédures

Une **procédure** est un **ensemble d'instructions** portant un **nom**. Pour définir une procédure, on utilise la syntaxe :

```
public static void nomprocedure()
{
  /*
        instructions
  */
}
```

Une procédure est une nouvelle instruction, il suffit pour l'exécuter d'utiliser son nom. Par exemple, pour **exécuter** (on dit aussi **appeler** ou **invoquer**) une procédure appelée *pr*, il suffit d'écrire *pr*();. Les deux programmes suivants font la même chose. Le premier est écrit sans procédure :

```
class MainClass
{
        public static void Main (string[] args)
        {
                Console.WriteLine("Bonjour !");
        }
  }
```

Et dans le deuxième, le `Console.WriteLine` est placé dans la procédure `afficheBonjour` et cette procédure est appelée depuis le `Main`.

```
class MainClass
{
        public static void afficheBonjour()
        {
                Console.WriteLine("Bonjour !");
        }

        public static void Main (string[] args)
        {
                afficheBonjour();
        }
}
```

Vous pouvez définir autant de procédures que vous le voulez et vous pouvez appeler des procédures depuis des procédures :

```
class MainClass
{
  public static void afficheBonjour()
  {
    Console.WriteLine("Bonjour,");
  }

  public static void afficheUn()
  {
    Console.WriteLine("1");
  }

  public static void afficheDeux()
```

```
  {
    Console.WriteLine("2");
  }

  public static void afficheUnEtDeux()
  {
    afficheUn();
    afficheDeux();
  }

  public static void afficheAuRevoir()
  {
    Console.WriteLine("Au revoir.");
  }

  public static void Main(String[] args)
  {
    afficheBonjour();
    afficheUnEtDeux();
    afficheAuRevoir();
  }
}
```

Ce programme affiche :

```
Bonjour,
1
2
Au revoir.
```

Regardez bien le programme suivant et essayez de déterminer ce qu'il affiche.

```
class MainClass
{

        public static void procedure1()
        {
          Console.WriteLine("debut procedure 1");
          Console.WriteLine("fin procedure 1");
        }

        public static void procedure2()
        {
          Console.WriteLine("debut procedure 2");
          procedure1();
          Console.WriteLine("fin procedure 2");
        }

        public static void procedure3()
        {
          Console.WriteLine("debut procedure 3");
          procedure1();
          procedure2();
          Console.WriteLine("fin procedure 3");
        }
```

```
        public static void Main(String[] args)
        {
          Console.WriteLine("debut main");
          procedure2();
          procedure3();
          Console.WriteLine("fin main");
        }
}
```

La réponse est

```
debut main
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
debut procedure 3
debut procedure 1
fin procedure 1
debut procedure 2
debut procedure 1
fin procedure 1
fin procedure 2
fin procedure 3
fin main
```

Vous remarquez au passage que `Main` est aussi une procédure. `Main` est exécutée automatiquement au lancement du programme.

1.8.2 Variables locales

Une procédure est un bloc d'instructions et est sujette aux mêmes règles que `Main`. Il est donc possible d'y déclarer des variables :

```
public static void nomprocedure()
{
  int i=0;
  Console.WriteLine(i);
}
```

Attention, ces variables ne sont accessibles que dans le corps de la procédure, cela signifie qu'elles naissent au moment de leur déclaration et qu'elles sont détruites une fois la dernière instruction de la procédure exécutée. C'est pour cela qu'on les apelle des **variables locales**. Une variable locale n'est **visible** qu'entre sa déclaration et l'accolade fermant la procédure. Par exemple, ce code est illégal :

```
class MainClass
{
  public static void maProcedure()
  {
    char a = b;
  }


  public static void Main(String[] args)
  {
    char b = 'k';
    Console.WriteLine(a + ", " + b);
```

```
  }
}
```

En effet, la variable b est déclarée dans la procédure `Main`, et n'est donc visible que dans cette même procédure. Son utilisation dans `maProcedure` est donc impossible. De même, la variable a est déclarée dans `maProcedure`, elle n'est pas visible dans le `Main`. Voici un exemple d'utilisation de variables locales :

```
class MainClass
{
  public static void unADix()
  {
    int i;
    for(i = 1 ; i <= 10 ; i++ )
    Console.WriteLine(i);
  }


  public static void Main(String[] args)
  {
    unADix();
  }
}
```

1.8.3 Passage de paramètres

Il est possible que la valeur d'une variable locale d'une procédure ne soit connue qu'au moment de l'appel de la procédure. Considérons le programme suivant :

```
public static void Main(String[] args)
{
  int i;
  Console.WriteLine("Veuillez saisir un entier : ");
  i = int.Parse(Console.ReadLine());
  /*
  Appel d'une procedure affichant la valeur de i.
  */
}
```

Comment définir et invoquer une procédure `afficheInt` permettant d'afficher cet entier saisi par l'utilisateur ? Vous conviendrez que la procédure suivante ne passera pas la compilation

```
public static void afficheInt()
{
  Console.WriteLine(i);
}
```

En effet, la variable i est déclarée dans le `Main`, elle n'est donc pas visible dans `afficheInt`. Pour y remédier, on définit `afficheInt` de la sorte :

```
public static void afficheInt(int i)
{
  Console.WriteLine(i);
}
```

i est alors appelé un **paramètre**, il s'agit d'une variable dont la valeur sera précisée lors de l'appel de la procédure. On peut aussi considérer que i est une valeur inconnue, et qu'elle est **initialisée** lors de l'invocation de la procédure. Pour initialiser la valeur d'un paramètre, on place cette valeur entre les

parenthèses lors de l'appel de la procédure, par exemple : `afficheInt`(4) lance l'exécution de la procédure `afficheInt` en initialisant la valeur de i à 4. On dit aussi que l'on **passe en paramètre** la valeur 4. La version correcte de notre programme est :

```
class MainClass
{
  public static void afficheInt(int i)
  {
    Console.WriteLine(i);
  }

  public static void Main(String[] args)
  {
    int i;
    Console.WriteLine("Veuillez saisir un entier : ");
    i = int.Parse(Console.ReadLine());
    afficheInt(i);
  }
}
```

Attention, notez bien que le i de `afficheInt` et le i du `Main` sont **deux variables différentes**, la seule chose qui les lie vient du fait que l'instruction `afficheInt(i)` initialise le i de `afficheInt` à la valeur du i du `Main`. Il serait tout à fait possible d'écrire :

```
class MainClass
{
  public static void afficheInt(int j)
  {
    Console.WriteLine(j);
  }

  public static void Main(String[] args)
  {
    int i;
    Console.WriteLine("Veuillez saisir un entier : ");
    i = int.Parse(Console.ReadLine());
    afficheInt(i);
  }
}
```

Dans cette nouvelle version, l'instruction `afficheInt(i)` initialise le j de `afficheInt` à la valeur du i du `Main`.

Il est possible de passer plusieurs valeurs en paramètre. Par exemple, la procédure suivante affiche la somme des deux valeurs passées en paramètre :

```
public static void afficheSomme(int a, int b)
{
  Console.WriteLine(a + b);
}
```

L'invocation d'une telle procédure se fait en initialisant les paramètres dans le même ordre et en séparant les valeurs par des virgules, par exemple `afficheSomme`(3, 5) invoque `afficheSomme` en initialisant a à 3 et b à 5. Vous devez initialiser tous les paramètres et vous devez placer les valeurs dans l'ordre. Récapitulons :

```
class MainClass
{
  public static void afficheSomme(int a, int b)
  {
    Console.WriteLine(a + b);
  }

  public static void Main(String[] args)
  {
    int i, j;
    Console.Write("Veuillez saisir deux entiers :\na = ");
    i = int.Parse(Console.ReadLine());
    Console.Write("b = ");
    j = int.Parse(Console.ReadLine());
    Console.Write("a + b = ");
    afficheSomme(i, j);
  }
}
```

La procédure ci-avant s'exécute de la façon suivante :

```
Veuillez saisir deux entiers :
a = 3
b = 5
a + b = 8
```

Lors de l'appel `afficheInt(r)` de la procédure **public static void** `afficheInt(int i)`, r est le **paramètre effectif** et i le **paramètre formel**. Notez bien que i et r sont deux variables distinctes. Par exemple, qu'affiche le programme suivant ?

```
class MainClass
{
  public static void incr(int v)
  {
    v++;
  }

  public static void Main(String[] args)
  {
    int i;
    i = 6;
    incr(i);
    Console.WriteLine(i);
  }
}
```

La variable v est initialisée à la valeur de i, mais i et v sont deux variables différentes. Modifier l'une n'affecte pas l'autre.

1.8.4 Les fonctions

Le principe

Nous avons vu qu'un sous-programme appelant peut communiquer des valeurs au sous-programme appelé. Mais est-il possible pour un sous-programme appelé de communiquer une valeur au sous-programme appelant ? La réponse est oui. Une **fonction** est un sous-programme qui communique une valeur au sous-programme appelant. Cette valeur s'appelle **valeur de retour**, ou **valeur retournée**.

Invocation

La syntaxe pour appeler une fonction est :

```
v = nomfonction(parametres);
```

L'instruction ci-dessus place dans la variable v la valeur retournée par la fonction `nomfonction` quand lui passe les paramètres `parametres`. Nous allons plus loin dans ce cours définir une fonction `carre` qui retourne le carré de valeur qui lui est passée en paramètre, alors l'instruction

```
v = carre(2);
```

placera dans v le carré de 2, à savoir 4. On définira aussi une fonction `somme` qui retourne la somme de ses paramètres, on placera donc la valeur $2 + 3$ dans v avec l'instruction

```
v = somme(2, 3);
```

Définition

On définit une fonction avec la syntaxe suivante :

```
public static typeValeurDeRetour nomFonction(listeParametres)
{
}
```

La fonction `carre` sera donc définie comme suit :

```
public static int carre(int i)
{
  /*
        instructions
  */
}
```

Une fonction ressemble beaucoup à une procédure. Vous remarquez que **void** est remplacé par **int**, **void** signifie aucune type de retour, une procédure est donc une fonction qui ne retourne rien. Un **int** est adapté pour représenter le carré d'un autre **int**, j'ai donc choisi comme **type de retour** le type **int**. Nous définirons la fonction `somme` comme suit :

```
public static int somme(int a, int b)
{
  /*
        instructions
  */
}
```

L'instruction servant à retourner une valeur est **return**. Cette instruction interrompt l'exécution de la fonction et retourne la valeur placée immédiatement après. Par exemple, la fonction suivante retourne toujours la valeur 1.

```
public static int un()
{
  return 1;
}
```

Lorsque l'on invoque cette fonction, par exemple

```
v = un();
```

La valeur 1, qui est retournée par `un` est affectée à v. On définit une fonction qui retourne le successeur de son paramètre :

```
public static int successeur(int i)
{
  return i + 1;
}
```

Cette fonction, si on lui passe la valeur 5 en paramètre, retourne 6. Par exemple, l'instruction

```
v = successeur(5);
```

affecte à v la valeur 6. Construisons Maintenant nos deux fonctions :

```
class MainClass
{
  public static int carre(int i)
  {
    return i * i ;
  }

  public static int somme(int a, int b)
  {
    return a + b ;
  }

  /*
  ...
  */
}
```

1.8.5 Passages de paramètre par référence

En C#, lorsque vous invoquez une fonction, toutes les valeurs des paramètres effectifs sont recopiés dans les paramètres formels. On dit dans ce cas que le passage de paramètre se fait **par valeur**. Vous ne pouvez donc, *a priori*, communiquer qu'une seule valeur au programme appelant. Par conséquent **seule la valeur de retour vous permettra de communiquer une valeur au programme appelant**.

Lorsque vous passez un tableau en paramètre, la valeur qui est recopiée dans le paramètre formel est l'adresse de ce tableau (l'adresse est une valeur scalaire). Par conséquent toute modification effectuée sur les éléments d'un tableau dont l'adresse est passée en paramètre par valeur sera repercutée sur le paramètre effectif (i.e. le tableau d'origine). Lorsque les modifications faites sur un paramètre formel dans un sous-programme sont repércutées sur le paramètre effectif, on a alors un **passage de paramètre par référence**.

```
class MainClass
{
  public static void initTab(int[] t)
  {
    int n = t.Length;
    for (int i = 0 ; i < n ; i++)
        t[i] = i+1;
  }

  public static int[] copieTab(int[] t)
  {
    int n = t.Length;
```

```
    int[] c = new int[n];
    for (int i = 0 ; i < n ; i++)
        c[i] = t[i];
    return c;
  }

  public static void Main(String[] args)
  {
    int[] t = new int[20];
    initTab(t);
    int [] c = copieTab(t);
    foreach(int x in c)
        Console.Write(x + " ");
    Console.WriteLine();
  }
}
```

Nous retiendrons donc les règles d'or suivantes :

- Les variables scalaires se passent en paramètre par valeur
- Les variables non scalaires se passent en paramètre par référence

1.9 Objets

Dans un langage de programmation, un **type** est
- Un ensemble de **valeurs**
- Des **opérations** sur ces valeurs

En plus des types primitifs, il est possible en C# de créer ses propres types. On appelle type construit un type non primitif, c'est-à-dire composé de types primitifs. Certains types construits sont fournis dans les bibliothèques du langage. Si ceux-là ne vous satisfont pas, vous avez la possibilité de créer vos propres types.

1.9.1 Création d'un type

Nous souhaitons créer un type `Point` dans R^2. Chaque variable de ce type aura deux attributs, une abscisse et une ordonnée. Le type point se compose donc à partir de deux types flottants. Un type construit s'appelle une **classe**. On le définit comme suit :

```
class Point
{
        public double abscisse;
        public double ordonnee;
}
```

Les deux attributs d'un objet de type `Point` s'appelle aussi des **champs**. Une fois définie cette classe, le type `Point` est une type comme les autres, il devient donc possible d'écrire

```
Point p, q;
```

Cette instruction déclare deux variables `p` et `q` de type `Point`, ces variables s'appellent des **objets**. Chacun de ces objets a deux attributs auxquels on accède avec la notation pointée. Par exemple, l'abscisse du point `p` est `p.abscisse` et son ordonnée est `p.ordonnee`.

Voyons un exemple d'utilisation de cette classe :

```
class Point
{
        public double abscisse;
        public double ordonnee;
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.ordonnee = 3;
                p.abscisse = 2;
                Console.WriteLine("p = (" + p.abscisse + ", " + p.ordonnee + ")");
        }
}
```

De la même façon que pour les tableaux, `p` n'est pas un point, mais une **variable contenant un point**. Par conséquent le **new** `Point()`; est indispensable !

1.9.2 L'instanciation

Revenons sur cette histoire de **new**. Quand vous déclarez une variable non primitive, elle ne contient rien. La valeur utilisée dans les langages de programmation objet pour spécifier la valeur *rien* est **null**.

Par conséquent, ce n'est pas la déclaration qui permet de créer un objet mais l'utilisation de l'instruction **new**.

Le problème que cela pose s'observe sur l'exemple suivant :

```
class Point
{
        public double abscisse;
        public double ordonnee;
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.ordonnee = 3;
                p.abscisse = 2;
                Point q = p;
                q.abscisse = 4;
                Console.WriteLine("p = (" + p.abscisse + ", " + p.ordonnee + ")");
        }
}
```

A votre avis, qu'affiche-t-il ? `q` est-il une copie de `p` ? Ou est-ce que `q` et `p` sont deux noms différents pour un même objet ?

Une variable de type `Point` n'est pas un point, mais l'adresse mémoire (identifiant, référence) d'un objet de type `Point`. Donc, `Point q = p;` crée une deuxième variable de type `Point`, mais qui contient la même adresse que `p`. Donc `p` et `q` référencent le même objet, toute modification sur l'objet référencé par `p` s'observera aussi sur l'objet référencé par `q`.

Lorsqu'un même objet porte plusieurs noms différents, on parle d'**aliasing**. Il convient d'être prudent avec les langages qui permettent l'aliasing, car cela peut engendrer des bugs très difficiles à trouver.

1.9.3 Les méthodes

Non contents d'avoir défini ainsi un ensemble de valeurs, nous souhaiterions définir un ensembe d'opérations sur ces valeurs. Nous allons pour ce faire nous servir de **méthodes**. Une méthode est un sous-programme propre à chaque objet. C'est-à-dire dont le contexte d'exécution est délimité par un objet. Par exemple,

```
class Point
{
    public double abscisse;
    public double ordonnee;

    public void presenteToi()
    {
        Console.WriteLine("Je suis un point, mes coordonnées sont ("
                          + abscisse + ", " + ordonnee + ")");
    }
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
```

```
                p.ordonnee = 3;
                p.abscisse = 2;
                p.presenteToi();
        }
}
```

On remarque qu'une méthode fonctionne comme un sous-programme à une différence près : il faut omettre le **static**.

La méthode `presenteToi` s'invoque à partir d'un objet de type `Point`. La syntaxe est `p.presenteToi()` où `p` est de type `Point`. `p` est alors le contexte de l'exécution de `presenteToi` et les champs auquel accèdera cette méthode seront ceux de l'objet `p`. Si par exemple, on écrit `q.presenteToi()`, c'est `q` qui servira de contexte à l'exécution de `presenteToi`. Lorsque l'on rédige une méthode, l'objet servant de contexte à l'exécution de la méthode est appelé l'**objet courant**.

Il est aussi possible de modifier les valeurs des attributs depuis une méthode :

```
class Point
{
        public double abscisse;
        public double ordonnee;

        public void initCoord(double a, double o)
        {
                abscisse = a;
                ordonnee = o;
        }

        public void presenteToi()
        {
                Console.WriteLine("Je suis un point, mes coordonnées sont ("
                        + abscisse + ", " + ordonnee + ")");
        }
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.initCoord(3, 2);
                p.presenteToi();
        }
}
```

Pour aller dans les applications tordues de la programmation objet, une méthode peut prendre en paramètre un autre objet :

```
class Point
{
        public double abscisse;
        public double ordonnee;

        public void initCoord(double a, double o)
        {
                abscisse = a;
                ordonnee = o;
        }
```

```
        public void copyCoord(Point autre)
        {
                abscisse = autre.abscisse;
                ordonnee = autre.ordonnee;
        }


        public void presenteToi()
        {
                Console.WriteLine("Je suis un point, mes coordonnées sont ("
                        + abscisse + ", " + ordonnee + ")");
        }
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.initCoord(3, 2);
                Point q = new Point();
                q.copyCoord(p);
                q.ordonnee ++;
                p.presenteToi();
                q.presenteToi();
        }
}
```

La méthode `initCoord` recopie les coordonnées d'un autre objet de type `Point` (`autre` en l'occurrence) à l'intérieur du point courant.

Vous avez aussi le droit de mettre au point des fonctions qui retournent un objet :

```
class Point
{
        public double abscisse;
        public double ordonnee;

        public void initCoord(double a, double o)
        {
                abscisse = a;
                ordonnee = o;
        }

        public void copyCoord(Point autre)
        {
                abscisse = autre.abscisse;
                ordonnee = autre.ordonnee;
        }

        public Point copyPoint()
        {
                Point res = new Point();
                res.ordonnee = ordonnee;
                res.abscisse = abscisse;
```

```
                return res;
        }

        public void presenteToi()
        {
                Console.WriteLine("Je suis un point, mes coordonnées sont ("
                   + abscisse + ", " + ordonnee + ")");
        }

}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.initCoord(3, 2);
                Point q = p.copyPoint();
                q.ordonnee ++;
                p.presenteToi();
                q.presenteToi();
        }
}
```

Le méthode `copyPoint` crée un point et le retourne après y avoir recopié les attributs du point courant. Donc, `p` et `q` sont deux points distincts.

1.9.4 Le mot-clé this

Dans toute méthode, vous disposez d'une référence vers l'objets servant de contexte à l'exécution de la méthode, cette référence s'appelle **this**.

```
class Point
{
        public double abscisse;
        public double ordonnee;

        public void initCoord(double abscisse, double ordonnee)
        {
                this.abscisse = abscisse;
                this.ordonnee = ordonnee;
        }

        public void copyCoord(Point autre)
        {
                abscisse = autre.abscisse;
                ordonnee = autre.ordonnee;
        }

        public Point copyPoint()
        {
                Point res = new Point();
                res.copyCoord(this);
                return res;
        }
```

```
        public void presenteToi()
        {
                Console.WriteLine("Je suis un point, mes coordonnées sont ("
                   + abscisse + ", " + ordonnee + ")");
        }
}

class MainClass
{
        public static void Main (string[] args)
        {
                Point p = new Point();
                p.initCoord(3, 2);
                Point q = p.copyPoint();
                q.ordonnee ++;
                p.presenteToi();
                q.presenteToi();
        }
}
```

Dans l'exemple ci-dessus, la méthode `initCoord` a été modifiée, les noms des paramètres formels sont les mêmes que les noms des champs. Le préfixe **this** permet d'utiliser des attributs et en l'absence de préfixe, C# choisit en priorité les variables locales. Dans la méthode `copyPoint`, le mot clé **this** est employé pour envoyer l'objet en courant en paramètre dans la méthode d'un autre point.

Chapitre 2

Exercices

2.1 Variables

2.1.1 Saisie et affichage

Exercice 1 - Saisie d'une chaîne

Ecrire un programme demandant à l'utilisateur de saisir son nom, puis affichant le nom saisi.

Exercice 2 - Saisie d'un entier

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique entière puis affichant cette valeur.

Exercice 3 - Permutation de 2 variables

Saisir deux variables et les permuter avant de les afficher.

Exercice 4 - Permutation de 4 valeurs

Ecrire un programme demandant à l'utilisateur de saisir 4 valeurs A, B, C, D et qui permute les variables de la façon suivante :

noms des variables	A	B	C	D
valeurs avant la permutation	1	2	3	4
valeurs après la permutation	3	4	1	2

Exercice 5 - Permutation de 5 valeurs

On considère la permutation qui modifie cinq valeurs de la façon suivante :

noms des variables	A	B	C	D	E
valeurs avant la permutation	1	2	3	4	5
valeurs après la permutation	4	3	5	1	2

Ecrire un programme demandant à l'utilisateur de saisir 5 valeurs que vous placerez dans des variables appelées A, B, C, D et E. Vous les permuterez ensuite de la façon décrite ci-dessus.

Exercice 6 - Permutation ultime

Même exercice avec :

noms des variables	A	B	C	D	E	F
valeurs avant la permutation	1	2	3	4	5	6
valeurs après la permutation	3	4	5	1	6	2

2.1.2 Entiers

Exercice 7 - Opération sur les entiers

Saisir deux variables entières, afficher leur somme et leur quotient.

2.1.3 Flottants

Exercice 8 - Saisie et affichage

Saisir une variable de type `float`, afficher sa valeur.

Exercice 9 - Moyenne arithmétique

Saisir 3 valeurs, afficher leur moyenne.

Exercice 10 - Surface du rectangle

Demander à l'utilisateur de saisir les longueurs et largeurs d'un rectangle, afficher sa surface.

2.1.4 Caractères

Exercice 11 - Prise en main

Affectez le caractère `'a'` à une variable de type `char`, affichez ce caractère ainsi que son code `UNICODE`.

Exercice 12 - Casse

Ecrivez un programme qui saisit un caractère miniscule et qui l'affiche en majuscule.

2.2 Opérateurs

2.2.1 Conversions

Exercice 1 - Successeur

Ecrivez un programme qui saisit un caractère et qui affiche son successeur dans la table des codes `UNICODE`.

Exercice 2 - Codes `UNICODE`

Quels sont les codes `UNICODE` des caractères '0', '1', ..., '9'

Exercice 3 - Moyennes arithmétique et géométrique

Demander à l'utilisateur de saisir deux valeurs a et b et type **float**. Afficher ensuite la différence entre la moyenne arithmétique $\frac{(a+b)}{2}$ et la moyenne géométrique $\sqrt{ab}$. Vous utiliserez l'instruction `Math.Sqrt(f)` qui donne la racine carrée du **double** `f`.

Exercice 4 - Cartons et camions

Nous souhaitons ranger des cartons pesant chacun k kilos dans un camion pouvant transporter M kilos de marchandises. Ecrire un programme C demandant à l'utilisateur de saisir M et k, que vous représenterez avec des nombres flottants, et affichant le nombre (entier) de cartons qu'il est possible de placer dans le camion.

2.2.2 Opérations sur les bits (difficiles)

Exercice 5 - Codage d'adresses IP

Une adresse `IP` est constituée de 4 valeurs de 0 à 255 séparées par des points, par exemple `192.168.0.1`, chacun de ces nombres peut se coder sur 1 octet. Comme certaines variables de type numérique occupent 4 octets en mémoire, il est possible de s'en servir pour stocker une adresse `IP` entière. Ecrivez un programme qui saisit dans 4 variables numériques d'un octet chaque valeur numérique constituant une adresse `IP`. Vous créerez ensuite une variable numérique de 4 octets dans laquelle vous placerez ces quatre valeurs. Ensuite vous mettrez en oeuvre le processus inverse : vous extrairez de cet entier les 4 nombres de l'adresse `IP` et les afficherez en les séparant par des points. Vous devriez ainsi retrouver les quatre nombres saisis par l'utilisateur...

Exercice 6 - Permutation circulaire des bits

Effectuez une permutation circulaire vers la droite des bits d'une variable `b` de type `byte`, faites de même vers la gauche.

Exercice 7 - Permutation de 2 octets

Permutez les deux octets d'une variable numérique à 2 octets saisie par l'utilisateur.

Exercice 8 - Inversion de l'ordre de 4 octets

Inversez l'ordre des octets d'une variable numérique de 4 octets saisie par l'utilisateur. Utilisez le code du programme sur les adresses `IP` pour tester votre programme.

2.2.3 Morceaux choisis (difficiles)

Exercice 9 - Pièces de monnaie

Nous disposons d'un nombre illimité de pièces de 0.5, 0.2, 0.1, 0.05, 0.02 et 0.01 euros. Nous souhaitons, étant donné une somme S, savoir avec quelles pièces la payer de sorte que le nombre de pièces utilisée soit minimal. Par exemple, la somme de 0.96 euros se paie avec une pièce de 0.5 euros, deux pièces de 0.2 euros, une pièce de 0.05 euros et une pièce de 0.01 euros.

1. Le fait que la solution donnée pour l'exemple est minimal est justifié par une idée plutôt intuitive. Expliquez ce principe sans excès de formalisme.
2. Ecrire un programme demandant à l'utilisateur de saisir une valeur comprise entre 0 et 0.99. Ensuite, affichez le détail des pièces à utiliser pour constituer la somme saisie avec un nombre minimal de pièces.

Exercice 10 - Modification du dernier bit

Modifiez le dernier bit d'une variable numérique `a` saisie par l'utilisateur.

Exercice 11 - Associativité de l'addition flottante

L'ensemble des flottants n'est pas associatif, cela signifie qu'il existe trois flottants a, b et c, tels que $(a + b) + c \neq a + (b + c)$. Trouvez de tels flottants et vérifiez-le dans un programme.

Exercice 12 - Permutation sans variable temporaire

Permutez deux variables `a` et `b` sans utiliser de variable temporaire.

2.3 Traitements conditionnels

2.3.1 Prise en main

Exercice 1 - Majorité

Saisir l'âge de l'utilisateur et lui dire s'il est majeur ou s'il est mineur.

Exercice 2 - Valeur Absolue

Saisir une valeur, afficher sa valeur absolue.

Exercice 3 - Admissions

Saisir une note, afficher "ajourné" si la note est inférieure à 8, "rattrapage" entre 8 et 10, "admis" dessus de 10.

Exercice 4 - Assurances

Une compagnie d'assurance effectue des remboursements sur lesquels est ponctionnée une franchise correspondant à 10% du montant à rembourser. Cependant, cette franchise ne doit pas excéder 4000 euros. Demander à l'utilisateur de saisir le montant des dommages, afficher ensuite le montant qui sera remboursé ainsi que la franchise.

Exercice 5 - Valeurs distinctes parmi 2

Afficher sur deux valeurs saisies le nombre de valeurs distinctes.

Exercice 6 - Plus petite valeur parmi 3

Afficher sur trois valeurs saisies la plus petite.

Exercice 7 - Recherche de doublons

Ecrire un algorithme qui demande à l'utilisateur de saisir trois valeurs et qui lui dit s'il s'y trouve un doublon.

Exercice 8 - Valeurs distinctes parmi 3

Afficher sur trois valeurs saisies le nombre de valeurs distinctes.

Exercice 9 - $ax + b = 0$

Saisir les coefficients a et b et afficher la solution de l'équation $ax + b = 0$.

Exercice 10 - $ax^2 + bx + c = 0$

Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$.

2.3.2 Switch

Exercice 11 - Calculatrice

Ecrire un programme demandant à l'utilisateur de saisir
- deux valeurs a et b, de type *int* ;
- un opérateur *op* de type *char*, vérifiez qu'il s'agit del'une des valeurs suivantes : $+, -, *, /$.

Puis affichez le résultat de l'opération a *op* b.

2.3.3 L'échiquier

On indice les cases d'un échiquer avec deux indices i et j variant tous deux de 1 à 8. La case (i, j) est sur la ligne i et la colonne j. Par convention, la case $(1, 1)$ est noire.

Exercice 12 - Couleurs

Ecrire un programme demandant à l'utilisateur de saisir les deux coordonnées i et j d'une case, et lui disant s'il s'agit d'une case blanche ou noire.

Exercice 13 - Cavaliers

Ecrire un programme demandant à l'utilisateur de saisir les coordonnées (i, j) d'une première case et les coordonnées (i', j') d'une deuxième case. Dites-lui ensuite s'il est possible de déplacer un cavalier de (i, j) à (i', j').

Exercice 14 - Autres pièces

Même exercice avec la tour, le fou, la dame et le roi. Utilisez un switch et présentez le programme de la sorte :

```
Quelle pièce souhaitez-vous déplacer ?
0 = cavalier
1 = Tour
2 = Fou
3 = Dame
4 = Roi
4
Coordonnées (i, j) de la position de départ :
i = 5
j = 6
Coordonnées (i', j') de la position d'arrivée :
i' = 6
j' = 7
Déplacement du roi de (5, 6) vers (6, 7) correct.
```

2.3.4 Heures et dates

Exercice 15 - Opérations sur les heures

Ecrire un programme qui demande à l'utilisateur de saisir une heure de début (heures + minutes) et une heure de fin (heures + minutes aussi). Ce programme doit ensuite calculer en heures + minutes le temps écoulé entre l'heure de début et l'heure de fin. Si l'utilisateur saisit $10h30$ et $12h15$, le programme doit lui afficher que le temps écoulé entre l'heure de début et celle de fin est $1h45$. On suppose que les deux heures se trouvent dans la même journée, si celle de début se trouve après celle de fin, un message d'erreur doit s'afficher. Lors la saisie des heures, séparez les heures des minutes en demandant à l'utilisateur de saisir :

- heures de début
- minutes de début
- heures de fin
- minutes de fin

Exercice 16 - Le jour d'après

Ecrire un programme permettant de saisir une date (jour, mois, année), et affichant la date du lendemain. Saisissez les trois données séparément (comme dans l'exercice précédent). Prenez garde aux nombre de jours que comporte chaque mois, et au fait que le mois de février comporte 29 jours les années bissextiles.

Allez sur `http://fr.wikipedia.org/wiki/Ann\%C3\%A9e_bissextilepourconnatrelesrglesexactes`, je vous avais dit que les années étaient bissextiles si et seulement si elles étaient divisible par 4, après vérification, j'ai constaté que c'était légèrement plus complexe. Je vous laisse vous documenter et retranscrire ces règles de la façon la plus simple possible.

2.3.5 Intervalles et rectangles

Exercice 17 - Intervalles bien formés

Demandez à l'utilisateur de saisir les deux bornes a et b d'un intervalle $[a, b]$. Contrôler les valeurs saisies.

Exercice 18 - Appartenance

Demandez-lui ensuite de saisir une valeur x, dites-lui si $x \in [a, b]$

Exercice 19 - Intersections

Demandez-lui ensuite de saisir les bornes d'un autre intervalle $[a', b']$. Contrôlez la saisie. Dites-lui ensuite si

- $[a, b] \subset [a', b']$
- $[a', b'] \subset [a, b]$
- $[a', b'] \cap [a, b] = \emptyset$.

Exercice 20 - Rectangles

Nous représenterons un rectangle R aux cotés parallèles aux axes des abscisses et ordonnées à l'aide des coordonnées de deux points diamétralement opposés, le point en haut à gauche, de coordonnées $(xHautGauche, yHautGauche)$, et le point en bas à droite, de coordonnées $(xBasDroite, yBasDroite)$. Demander à l'utilisateur de saisir ces 4 valeurs, contrôlez la saisie.

Exercice 21 - Appartenance

Demandez à l'utilisateur de saisir les 2 coordonnées d'un point (x, y) et dites à l'utilisateur si ce point se trouve dans le rectangle R.

Exercice 22 - Intersection

Demandez à l'utilisateur de saisir les 4 valeurs

$$(xHautGauche', yHautGauche', xBasDroite', yBasDroite')$$

permettant de spécifier un deuxième rectangle R'. Précisez ensuite si

- $R \subset R'$
- $R' \subset R$
- $R \cap R' = \emptyset$

2.4 Boucles

2.4.1 Compréhension

Exercice 1

Qu'affiche le programme suivant ?

```
using System;

namespace tests
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        int a = 1, b = 0, n = 5;
                        while(a <= n)
                                b += a++;
                        Console.WriteLine(a + ", " + b);
                }
        }
}
```

Exercice 2

Qu'affiche le programme suivant ?

```
using System;

namespace tests
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        int a, b, c = 0, d, m=3, n=4;
                        for (a = 0 ; a < m ; a++)
                        {
                                d = 0;
                                for(b = 0 ; b < n ; b++)
                                        d+=b;
                                c += d;
                        }
                        Console.WriteLine(a + ", " + b + ", " + c + ", " + d + ".");
                }
        }
}
```

Exercice 3

Qu'affiche le programme suivant ?

```
using System;

namespace tests
```

```
{
        class MainClass
        {
                public static void Main (string[] args)
                {
                        int a, b, c, d;
                        a = 1; b = 2;
                        c = a/b;
                        d = (a==b)?3:4;
                        Console.WriteLine(c + ", " + d + ".");
                        a = ++b;
                        b %= 3;
                        Console.WriteLine(a + ", " + b + ".");
                        b = 1;
                        for(a = 0 ; a <= 10 ; a++)
                                c = ++b;
                        Console.WriteLine(a + ", " + b + ", " + c + ", " + d + ".");
                }
        }
}
```

2.4.2 Utilisation de toutes les boucles

Les exercices suivants seront rédigés avec les trois types de boucle : tant que, répéter jusqu'à et pour.

Exercice 4 - Compte à rebours

Écrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant toutes les valeurs $n, n-1, \ldots, 2, 1, 0$.

Exercice 5 - Factorielle

Ecrire un programme calculant la factorielle (factorielle $n = n! = 1 \times 2 \times \ldots \times n$ et $0! = 1$) d'un nombre saisi par l'utilisateur.

2.4.3 Choix de la boucle la plus appropriée

Pour les exercices suivants, vous choisirez la boucle la plus simple et la plus lisible.

Exercice 6 - Table de multiplication

Écrire un programme affichant la table de multiplication d'un nombre saisi par l'utilisateur.

Exercice 7 - Tables de multiplications

Écrire un programme affichant les tables de multiplications des nombres de 1 à 10 dans un tableau à deux entrées.

Exercice 8 - Puissance

Écrire un programme demandant à l'utilisateur de saisir deux valeurs numériques b et n (vérifier que n est positif) et affichant la valeur b^n.

Exercice 9 - Joli carré

Écrire un programme qui saisit une valeur n et qui affiche le carré suivant ($n = 5$ dans l'exemple) :

```
n = 5
X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X

X  X  X  X  X
```

2.4.4 Morceaux choisis

Exercice 10 - Approximation de 2 par une série

On approche le nombre 2 à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{2^i}$. Effectuer cette approximation en calculant un grand nombre de termes de cette série. L'approximation est-elle de bonne qualité ?

Exercice 11 - Approximation de e par une série

Mêmes questions qu'à l'exercice précédent en e à l'aide de la série $\sum_{i=0}^{+\infty} \frac{1}{i!}$.

Exercice 12 - Approximation de e^x par une série

Calculer une approximation de e^x à l'aide de la série $e^x = \sum_{i=0}^{+\infty} \frac{x^i}{i!}$.

Exercice 13 - Conversion d'entiers en binaire

Écrire un programme qui affiche un `unsigned short` en binaire. Vous utiliserez l'instruction `sizeof(unsigned short)`, qui donne en octets la taille de la représentation en mémoire d'un `unsigned short`.

Exercice 14 - Conversion de décimales en binaire

Écrire un programme qui affiche les décimales d'un `double` en binaire.

Exercice 15 - Inversion de l'ordre des bits

Écrire un programme qui saisit une valeur de type `unsigned short` et qui inverse l'ordre des bits. Vous testerez ce programme en utilisant le précédent.

Exercice 16 - Racine carrée par dichotomie

Écrire un algorithme demandant à l'utilisateur de saisir deux valeurs numériques x et p et affichant $\sqrt{x}$ avec une précision p. On utilisera une méthode par dichotomie : à la k-ème itération, on cherche x dans l'intervalle $[min, sup]$, on calcule le milieu m de cet intervalle (à vous de trouver comment le calculer). Si cet intervalle est suffisament petit (à vous de trouver quel critère utiliser), afficher m. Sinon, vérifiez si $\sqrt{x}$ se trouve dans $[inf, m]$ ou dans $[m, sup]$, et modifiez les variables inf et sup en conséquence. Par exemple, calculons la racine carrée de 10 avec une précision 0.5,

- Commençons par la chercher dans $[0, 10]$, on a $m = 5$, comme $5^2 > 10$, alors $5 > \sqrt{10}$, donc $\sqrt{10}$ se trouve dans l'intervalle $[0, 5]$.

- On recommence, $m = 2.5$, comme $\frac{5}{2}^2 = \frac{25}{4} < 10$, alors $\frac{5}{2} < \sqrt{10}$, on poursuit la recherche dans $[\frac{5}{2}, 5]$
- On a $m = 3.75$, comme $3.75^2 > 10$, alors $3.75 > \sqrt{10}$ et $\sqrt{10} \in [2.5, 3.75]$
- On a $m = 3.125$, comme $3.125^2 < 10$, alors $3.125 < \sqrt{10}$ et $\sqrt{10} \in [3.125, 3.75]$
- Comme l'étendue de l'intervalle $[3.125, 3.75]$ est inférieure 2×0.5, alors $m = 3.4375$ est une approximation à 0.5 près de $\sqrt{10}$.

2.4.5 Extension de la calculatrice

Une calculatrice de poche prend de façon alternée la saisie d'un opérateur et d'une opérande. Si l'utilisateur saisit 3, + et 2, cette calculatrice affiche 5, l'utilisateur a ensuite la possibilité de se servir de 5 comme d'une opérande gauche dans un calcul ultérieur. Si l'utilisateur saisit par la suite $*$ et 4, la calculatrice affiche 20. La saisie de la touche = met fin au calcul et affiche un résultat final.

Exercice 17 - Calculatrice de poche

Implémentez le comportement décrit ci-dessus.

Exercice 18 - Puissance

Ajoutez l'opérateur $ qui calcule a^b, vous vous restreindrez à des valeurs de b entières et positives.

Exercice 19 - Opérations unaires

Ajoutez les opérations unaires racine carrée et factorielle.

2.4.6 Révisions (SISR)

Exercice 20

1. Écrire un programme saisissant un entier et l'affichant.
2. Précisez ensuite à l'utilisateur si ce nombre est positif ou négatif.
3. Affichez ensuite toutes les valeurs entre ce nombre et 0.
4. Affichez ensuite la somme des nombres affichés.

Exercice 21

1. Écrire un programme saisissant deux entiers i et j et disant lequel est supérieur à l'autre.
2. Dans le cas où i est plus petit que j, affichez toutes les valeurs se trouvant entre i et j.
3. Même question mais en n'affichant que les valeurs paires.
4. Même question en donnant à la fin le produit des nombres affichés.

Exercice 22 - C+C-

1. Programmez un "C'est plus, c'est moins".
2. Inversez les rôles avec l'ordinateur, c'est à lui de deviner le nombre que vous avez choisi.

2.5 Chaînes de caractères

2.5.1 Prise en main

Question 1 - Affichage

Créer une chaîne de caractères contenant la valeur "Les framboises sont perchées sur le tabouret de mon grand-pere." et affichez-la caractère par caractère.

Question 2 - Extraction

Ecrire un programme saisissant une chaîne de caractère t, deux indices i et j et recopiant dans une deuxième chaîne t' la tranche $[t_i, \ldots, t_j]$. Vous construirez la deuxième chaîne par concaténations successives (sans `Substring` ni `System.Text.StringBuilder`).

Question 3 - Extraction sans concaténation

Ecrire un programme saisissant une chaîne de caractère t, deux indices i et j et recopiant dans une deuxième chaîne t' la tranche $[t_i, \ldots, t_j]$. Vous construirez la deuxième chaîne en utilisant la fonction `insert(indice, caractère)` de `System.Text.StringBuilder`.

Question 4 - Substitution

Ecrire un programme saisissant une chaîne de caractère t, deux caractères a et b et substituant des a à toutes les occurrences de b. Vous utiliserez `s.Replace(a, b)`, qui crée une copie de s dans laquelle tous les a ont été remplacés par des b.

Question 5 - Substitution sans Replace

Ecrire un programme saisissant une `Stringbuilder` t, deux caractères a et b et modifiant t pour substituer des a à toutes les occurrences de b. Vous n'utiliserez donc pas `Replace` !

2.5.2 Morceaux choisis

Question 6 - Extensions

Ecrire un programme saisissant un nom de fichier et affichant séparément le nom du fichier et l'extension. Dans le cas où plusieurs extensions sont concaténées (par exemple : `langageC.tar.gz`), vous n'afficherez que la dernière extension (donc `.gz`).

Question 7 - Expressions arithmétiques

Ecrire un programme saisissant une expression arithmétique totalement parenthésée, (par exemple $3 + 4$, $((3 - 2) + (7/3))$) et disant à l'utilisateur si l'expression est correctement parenthésée.

Question 8 - Le pendu

Ecrire un programme saisissant un mot et demandant à un deuxième utilisateur de deviner le mot en un nombre d'essai fini. Pour ce faire l'utilisateur saisit une lettre et le programme lui affiche les occurrences de cette lettre dans le mot à trouver.

2.6 Tableaux

2.6.1 Exercices de compréhension

Qu'affichent les programmes suivants ?

Exercice 1

```
char[] c = new char[4];
c[0] = 'a';
c[3] = 'J';
c[2] = 'k';
c[1] = 'R';
for(int k = 0 ; k < 4 ; k++)
        Console.WriteLine(c[k]);
for(int k = 0 ; k < 4 ; k++)
        c[k]++;
foreach(char i in c)
       Console.WriteLine(i);
```

Exercice 2

```
int[] k;
k = new int[10];
k[0] = 1;
for(int i = 1 ; i < 10 ; i++)
        k[i] = 0;
for(int j = 1 ; j <= 3 ; j++)
        for(int i = 1 ; i < 10 ; i++)
                k[i] += k[i - 1];
foreach(int i in k)
        Console.WriteLine(i);
```

Exercice 3

```
int[] k;
k = new int[10];
k[0] = 1;
k[1] = 1;
for(int i = 2 ; i < 10 ; i++)
        k[i] = 0;
for(int j = 1 ; j <= 3 ; j++)
        for(int i = 1 ; i < 10 ; i++)
                k[i] += k[i - 1];
foreach(int p in k)
        Console.WriteLine(p);
```

2.6.2 Prise en main

Exercice 4 - Initialisation et affichage

Ecrire un programme plaçant dans un tableau `int[] T;` les valeurs $1, 2, \ldots, 10$, puis affichant ce tableau. Vous initialiserez le tableau à la déclaration.

Exercice 5 - Initialisation avec une boucle

Même exercice en initialisant le tableau avec une boucle.

Exercice 6 - Somme

Affichez la somme des n éléments du tableau T.

Exercice 7 - Recherche

Demandez à l'utilisateur de saisir un *int* et dites-lui si ce nombre se trouve dans T.

2.6.3 Indices

Exercice 8 - Permutation circulaire

Placez dans un deuxième tableau la permutation circulaire vers la droite des éléments de T.

Exercice 9 - Permutation circulaire sans deuxième tableau

Même exercice mais sans utiliser de deuxième tableau.

Exercice 10 - Miroir

Inversez l'ordre des éléments de T sans utiliser de deuxième tableau.

2.6.4 Recherche séquentielle

Exercice 11 - Modification du tableau

Etendez le tableau `T` à 20 éléments. Placez dans `T[i]` le reste modulo 17 de i^2.

Exercice 12 - Min/max

Affichez les valeurs du plus petit et du plus grand élément de `T`.

Exercice 13 - Recherche séquentielle

Demandez à l'utilisateur de saisir une valeur `x` et donnez-lui la liste des indices `i` tels que `T[i]` a la valeur `x`.

Exercice 14 - Recherche séquentielle avec stockage des indices

Même exercice que précédemment, mais vous en affichant `La valeur ... se trouve aux indices suivants : ...` si `x` se trouve dans `T`, et `La valeur ... n'a pas été trouvée` si `x` ne se trouve pas dans `T`. Vous utiliserez un tableau `Q` dans lequel vous stockerez les indices auxquels `x` aura été trouvé dans `T`.

2.6.5 Morceaux choisis

Exercice 15 - Pièces de monnaie

Reprenez l'exercice sur les pièces de monnaie en utilisant deux tableaux, un pour stocker les valeurs des pièces dans l'ordre décroissant, l'autre pour stocker le nombre de chaque pièce.

Exercice 16 - Impôt sur le revenu

Refaites le programme de calcul de l'impôt sur le revenu en utilisant des tableaux.

Exercice 17 - Recherche de la tranche minimale en 3 boucles

Une tranche est délimitée par deux indices i et j tels que $i \leq j$, la valeur d'une tranche est $t_i + \ldots + t_j$. Ecrire un programme de recherche de la plus petite tranche d'un tableau, vous utiliserez trois boucles imbriquées. Vous testerez votre algorithme sur un tableau T à 20 éléments aléatoires de signes quelconques.

Exercice 18 - Recherche de la tranche minimale en 2 boucles (difficile)

Même exercice mais en utilisant deux boucles imbriquées. Vous évaluerez $t_i + \ldots + t_{j+1}$ en calculant $(t_i + \ldots + t_j) + t_{j+1}$.

Exercice 19 - Recherche de la tranche minimale en 1 boucle (très difficile)

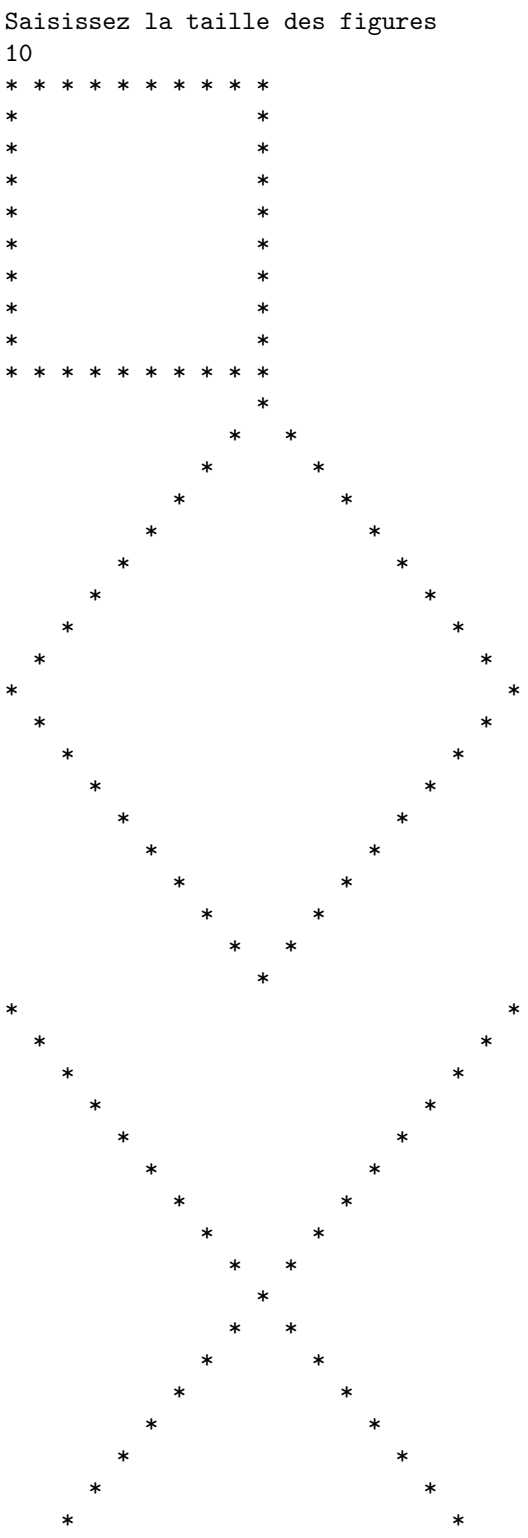
Même exercice mais en utilisant une seule boucle. Vous trouverez une relation simple entre :
- la plus petite tranche de $t_0, \ldots, t_j$
- la plus petite tranche de $t_0, \ldots, t_j$ contenant t_j
- la plus petite tranche de $t_0, \ldots, t_{j+1}$
- la plus petite tranche de $t_0, \ldots, t_{j+1}$ contenant t_{j+1}

Bon courage !

2.7 Sous-programmes

2.7.1 Géométrie

Ecrire un programme demandant à l'utilisateur de saisir une valeur numérique positive n et affichant un carré, une croix et un losange, tous de coté n. Par exemple, si $n = 10$, l'exécution donne



```
  *                                   *
*                                       *
```

Vous définirez des sous-programmes de quelques lignes et au plus deux niveaux d'imbrication. Vous ferez attention à ne jamais écrire deux fois les mêmes instructions. Pour ce faire, complétez le code source suivant :

```
using System;

namespace tests
{
        class MainClass
        {


                /*
                  Affiche le caractere c
                 */
                public static void afficheCaractere(char c)
                {
                }

                /****************************************/

                /*
                  affiche n fois le caractere c, ne revient pas a la ligne
                  apres le dernier caractere.
                */
                public static void ligneSansReturn(int n, char c)
                {
                }

                /****************************************/

                /*
                  affiche n fois le caractere c, revient a la ligne apres
                  le dernier caractere.
                */
                public static void ligneAvecReturn(int n, char c)
                {
                }

                /****************************************/

                /*
                  Affiche n espaces.
                */
                public static void espaces(int n)
                {
                }

                /****************************************/

                /*
                  Affiche le caractere c a la colonne i,
                  ne revient pas a la ligne apres.
                */
                public static void unCaractereSansReturn(int i, char c)
                {
                }

                /****************************************/

                /*
                  Affiche le caractere c a la colonne i,
                  revient a la ligne apres.
                */
                public static void unCaractereAvecReturn(int i, char c)
                {
                }

                /****************************************/

                /*
```

```
                    Affiche le caractere c aux colonnes i et j,
                    revient a la ligne apres.
                */

                public static void deuxCaracteres(int i, int j, char c)
                {
                }

                /****************************************/

                /*
                    Affiche un carre de cote n.
                */

                public static void carre(int n)
                {
                }

                /****************************************/

                /*
                    Affiche un chapeau dont la pointe - non affichee - est
                    sur la colonne centre, avec les caracteres c.
                */

                public static void chapeau(int centre, char c)
                {
                }

                /****************************************/

                /*
                    Affiche un chapeau a l'envers avec des caracteres c,
                    la pointe - non affichee - est a la colonne centre
                */

                public static void chapeauInverse(int centre, char c)
                {
                }

                /****************************************/

                /*
                    Affiche un losange de cote n.
                */

                public static void losange(int n)
                {
                }

                /****************************************/

                /*
                    Affiche une croix de cote n
                */

                public static void croix(int n)
                {
                }

                /****************************************/

                public static void Main(string[] args)
                {
                        int taille;
                        Console.WriteLine("Saisissez la taille des figures : ");
                        taille = int.Parse(Console.ReadLine());
                        carre(taille);
                        losange(taille);
                        croix(taille);
                }
        }
}
```

2.7.2 Arithmétique

Exercice 1 - chiffres et nombres

1. Ecrire la fonction **public static int** `unites`(**int** `n`) retournant le chiffre des unités du nombre n.

2. Ecrire la fonction **public static int** `dizaines`(**int** `n`) retournant le chiffre des dizaines du nombre n.
3. Ecrire la fonction **public static int** `extrait`(**int** `n`, **int** `p`) retourant le p-ème chiffre de représentation décimale de n en partant des unités.
4. Ecrire la fonction **public static int** `nbChiffres`(**int** `n`) retournant le nombre de chiffres que comporte la représentation décimale de n.
5. Ecrire la fonction **public static int** `sommeChiffres`(**int** `n`) retournant la somme des chiffres de n.

Exercice 2 - Nombres amis

Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.

1. Ecrire une fonction **public static int** `sommeDiviseursStricts`(**int** `n`), elle doit renvoyer la somme des diviseurs stricts de n.
2. Ecrire une fonction **public static bool** `sontAmis`(**int** `a`, **int** `b`), elle doit renvoyer 1 si a et b sont amis, 0 sinon.

Exercice 3 - Nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts. Par exemple, 6 a pour diviseurs stricts 1, 2 et 3, comme $1 + 2 + 3 = 6$, alors 6 est parfait.

1. Est-ce que 18 est parfait ?
2. Est-ce que 28 est parfait ?
3. Que dire d'un nombre ami avec lui-même ?
4. Ecrire la fonction **public static bool** `estParfait`(**int** `n`), elle doit retourner 1 si n est un nombre parfait, 0 sinon.

Exercice 4 - Nombres de Kaprekar

Un nombre n est un nombre de Kaprekar en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$. $45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de Kaprekar. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, je l'ai juste placé pour éviter toute confusion), alors 4879 est encore un nombre de Kaprekar.

1. Est-ce que 9 est un nombre de Kaprekar ?
2. Ecrire la fonction **public static int** `sommeParties`(**int** `n`, **int** `p`) qui découpe n est deux nombres dont le deuxième comporte p chiffres, et qui retourne leur somme. Par exemple,

$$sommeParties(12540, 2) = 125 + 40 = 165$$

3. Ecrire la fonction **public static bool** `estKaprekar`(**int** `n`)

2.7.3 Passage de tableaux en paramètre

Exercice 5 - Somme

Ecrire une fonction **public static int** `somme`(**int**[] `T`) retournant la somme des éléments de T.

Exercice 6 - Minimum

Ecrire une fonction **public static int** `min`(**int**[] `T`) retournant la valeur du plus petit élément de T.

Exercice 7 - Recherche

Ecrire une fonction **public static bool** `existe`(**int**[] `T`, **int** `k`) retournant **true** si et seulement si k est un des éléments de T.

Exercice 8 - Somme des éléments pairs

Ecrivez le corps de la fonction **public static int** `sommePairs`(**int**[] `T`), $sommePairs(T, n)$ retourne la somme des éléments pairs de T. N'oubliez pas que $a\%b$ est le reste de la division entière de a par b.

Exercice 9 - Vérification

Ecrivez le corps de la fonction **public static bool** `estTrie`(**int**[] `T`), $estTrie(T, n)$ retourne vrai si et seulement si les éléments de T sont triés dans l'ordre croissant.

Exercice 10 - Permutation circulaire

Ecrire une fonction **public static void** `permutation`(**int**[] `T`) effectuant une permutation circulaire vers la droite des éléments de T.

Exercice 11 - Miroir

Ecrire une fonction **public static void** `miroir`(**int**[] `T` inversant l'ordre des éléments de T.

2.7.4 Décomposition en facteurs premiers

On rappelle qu'un nombre est premier s'il n'est divisible que par 1 et par lui-même. Par convention, 1 n'est pas premier.

Exercice 12

Écrivez une fonction **public static bool** `estPremier`(**int** `x`, **int**[] `premiers`, **int** `k`) retournant vrai si et seulement si x est premier. Vous vérifierez la primarité de x en examinant les restes des divisions de x par les k premiers éléments de *premiers*. On suppose que k est toujours supérieur ou égal à 1.

Exercice 13

Modifiez la fonction précédente en tenant compte du fait que si aucun diviseur premier de x inférieur à $\sqrt{x}$ n'a été trouvé, alors x est premier

Exercice 14

Écrivez une fonction **public static int** [] `trouvePremiers`(**int** `n`) retournant un tableau contenant les n premiers nombres premiers.

Exercice 15

Écrivez une fonction **public static int**[] `decompose`(**int** `x`, **int**[] `premiers`) retournant un tableau contenant la décomposition en facteurs premiers du nombre x, sachant que T contient les n premiers nombres premiers. Par exemple, si $x = 108108$, alors on décompose n en produit de facteurs premiers de la sorte

$$108108 = 2 * 2 * 3 * 3 * 3 * 7 * 11 * 13 = 2^2 * 3^3 * 5^0 * 7^1 * 11^1 * 13^1 * 17^0 * 19^0 * \ldots * Z^0$$

(où Z est le n-ième nombre premier). On représente donc x de façon unique par le tableau à n éléments suivant :

$$\{2, 3, 0, 1, 1, 1, 0, 0, 0, \ldots, 0\}$$

Exercice 16

Écrivez une fonction **public static int** `recompose`(**int**[] `decomposition`, **int** [] `premiers`) effectuant l'opération réciproque de celle décrite ci-dessus.

Exercice 17

Écrivez une fonction **public static int**[] `pgcd`(**int**[] `T`, **int**[] `K`) prenant en paramètre les décompositions en facteurs premiers T et K de deux nombres, retournant la décomposition en facteurs premiers du plus grand commun diviseur de ces deux nombres.

Exercice 18

Écrivez une fonction **public static int** `pgcd`(**int** `i`, **int** `j`) prenant en paramètres deux nombres i et j, et combinant les fonctions précédentes pour retourner le *pgcd* de i et j. Vous poserez n suffisamment grand pour le calcul puisse fonctionner correctement.

2.8 Objets

2.8.1 Création d'une classe

Exercice 1 - La classe `Rationnel`

Créez une classe `Rationnel` contenant un numérateur et un dénominateur tous deux de type **long**. Instanciez deux rationnels a et b et initialisez-les aux valeurs respectives $\frac{1}{2}$ et $\frac{4}{3}$. Affichez ensuite les valeurs de ces champs.

2.8.2 Méthodes

Exercice 2 - Opérations sur les Rationnels

Ajoutez à la classe `Rationnel` les méthodes suivantes :

1. **public** `String toString()`, retourne une représentation du rationnel courant sous forme de chaîne de caractères.
2. **public static** `Rationnel create(long numerateur, long denominateur)`, retourne le rationnel *numerateur/de*
3. **public** `Rationnel copy()`, retourne une copie du rationnel courant.
4. **public** `Rationnel opposite()`, retourne l'opposé du rationnel courant.
5. **public** `Rationnel inverse()`, retourne l'inverse du rationnel courant.
6. **public void** `reduce()`, met le rationnel sous forme de fraction irréductible. Vous utiliserez l'algorithme d'Euclide calculant le plus grand commun diviseur.
7. **public bool** `isPositive()`, retourne **true** si et seulement si le rationnel courant est strictement positif.
8. **public** `Rationnel add(Rationnel other)`, retourne la somme du rationnel courant et du rationnel `other`.
9. **public void** `addTo(Rationnel other)`, additionne le rationnel `other` au rationnel courant.
10. **public** `Rationnel sub(Rationnel other)`, retourne la soustraction du rationnel courant et du rationnel `other`.
11. **public** `Rationnel multiply(Rationnel other)`, retourne le produit du rationnel courant avec le rationnel `others`.
12. **public** `Rationnel divide(Rationnel other)`, retourne le quotient du rationnel courant avec le rationnel `others`.
13. **public bool** `equals(Rationnel other)`, retourne vrai si et seulement si **this** et `others` sont égaux. Attention, 1/2 est égal à 2/4.
14. **public int** `compareTo(Rationnel other)`, retourne 0 si le rationnel courant est égal au rationnel `other`, -1 si le rationnel courant est inférieur à `other`, 1 dans le cas contraire.