



Cégep de Saint-Hyacinthe
Département d'informatique

Programmation orientée objet

420-2DP-HY

(3-3-3)

Révision des notions antérieures (survol)

(Version 1.2)

2 heures

Préparé par

Martin Lalancette

Comprendre les éléments suivants:

- Commentaires
- Constantes et variables
- Transtypage et conversion
- Opérateurs
- Structures alternatives et opérateurs logiques
- Structures répétitives
- Les tableaux
- Notions d'objets
- La liste générique

Table des matières

Introduction.....	4
Constantes.....	4
Indentation.....	4
Commentaires ou organisation du code	5
Les variables de type valeur	6
Les limites	6
Initialisation vs Affectation.....	7
La portée d’une variable.....	8
Le transtypage (« casting » en anglais)	9
Conversion de type <i>string</i> vers numérique	9
Les variables de type string	10
Accéder à un caractère de la chaîne	10
Les opérateurs.....	10
Structures alternatives (Conditions).....	12
Opérateurs logiques conditionnels	13
Méthodes et fonctions	14
Notions de compteur	15
Structures répétitives (Boucles)	15
Les fichiers de type texte.....	16
<i>namespace</i> à utiliser.....	16
Les classes File et Directory.....	16
La classe StreamReader.....	17
La classe StreamWriter.....	19
Les tableaux.....	21
Définition d’un tableau 1D	21
Tableau à deux dimensions (2D)	22
Méthode GetLength	23
Les index.....	23
La liste générique	23
Espace de nom	23

Déclaration d'une liste	23
Propriétés d'une liste	24
Quelques méthodes de base	24
Exercices	25
Bibliographie.....	29

Introduction

Cette séquence a pour but de vous faire réviser les notions de base nécessaires à la programmation orientée objet. Pour bien suivre les instructions, une préparation de base s'impose. Il est important de créer un répertoire de travail (sur votre C: ou clé USB). Voici une suggestion d'arborescence:

Constantes

Une constante nommée est **associée à une valeur pour toute la durée de l'application**. Sa valeur ne peut changer. En C#, c'est le mot **const** qui permet de définir une constante. Les règles suivantes pour l'attribution d'un nom :

1. Tous les mots sont en majuscules
2. Les mots sont séparés par des soulignements (_)

```
// -----
// Déclaration des constantes.
// -----
const byte MAX_NOMBRE_ITERATION = 5;           // Le nombre d'itérations possibles
const string COMMANDE_ARRÊT = "ARRÊT";        // Mot clé qui permet l'arrêt du traitement
const short DIMENSION_TABLEAU = 1000;         // Limite maximale de la dimension d'un
tableau
const bool FUMEUR = true;                      // Valeur représentant un fumeur
const bool NON_FUMEUR = false;                 // Valeur représentant un non-fumeur
```

Indentation

À chaque fois qu'un nouveau bloc d'instructions est débuté, et qu'il est subordonné à une ligne précédente (*if*, *while*, etc.), alors il faut le mettre en retrait par rapport à la ligne précédente. Par exemple, dans le *if* imbriqué suivant, il y a deux retraits, le second plus profond que le premier.

```
if (a > b)
{
    Temporaire = a;
    b = Temporaire;
    if (b > 3)
    {
        Console.WriteLine("b supérieur à 3 !");
    }
}
```

Le retrait doit être de 4 caractères → touche TABULATION.

Commentaires ou organisation du code

Type	Description
Commentaire De ligne	Pour inscrire un commentaire sur une ligne seulement, il suffit d'utiliser <code>//</code> au début et par la suite inscrire le commentaire. Exemple : <code>// Voici un exemple de déclaration d'une variable</code> <code>int iMois = 2; // Variable représentant les mois.</code>
Commentaire En bloc	Pour inscrire un commentaire sur plusieurs lignes (dit « en bloc »), il faut utiliser les caractères suivants : <code>/*</code> (pour l'ouverture du commentaire) et <code>*/</code> (pour la fermeture). Exemple : <code>/*</code> <code>void EffectuerOpération()</code> <code>{</code> <code>}</code> <code>*/</code>
Commentaire Sommaire (XML)	Dans Visual Studio et taper <code>///</code> (3 fois), alors les balises <code><summary></code> et <code></summary></code> seront autogénérées pour permettre la saisie d'une description. Exemple : <code>/// <summary></code> <code>/// Auteur : Martin Lalancette</code> <code>/// Date: 2012-10-04</code> <code>/// Description: Cette classe permet de créer un programme de</code> <code>/// type console qui servira à faire divers</code> <code>/// calculs sur des rectangles.</code> <code>/// </summary></code>
Organisation #region #endregion	Vous permet de spécifier un bloc de code que vous pouvez développer ou réduire lors de l'utilisation de la fonctionnalité Mode Plan de l'éditeur de code de Visual Studio.

Les variables de type valeur

Les limites

Le C# offre plusieurs types de bases en fonction d'un système d'exploitation 32 bits, voici les plus courants :

Type de données	Taille	Limites de valeur
byte sbyte (signed)	1 octet	0 à 255 -128 à 127
short ushort (unsigned)	2 octets	-32 768 à 32 767 0 à 65 535
int uint (unsigned)	4 octets	-2 147 483 648 à 2 147 483 647 0 à 4 294 967 295
long ulong (unsigned)	8 octets	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 0 à 18 446 744 073 709 551 615
float Précision : 7-8 digits	4 octets	-3.4×10^{38} à $+3.4 \times 10^{38}$ Exemples : float fValeur = 3.1234567f; float fValeur = 34.123456f;
double Précision : 15-16 digits	8 octets	$\pm 5.0 \times 10^{-324}$ à $\pm 1.7 \times 10^{308}$ // d (facultatif) double dValeur = 3.123456789012345d; double dValeur = 34.12345678901234d;
decimal Précision : 28-29 digits	16 octets	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$
bool	1 octet	Valeurs possibles : true et false (défaut)
char	2 octets	U+0000 to U+ffff Unicode (framework : System.Char). Afin de supporter tous les caractères de toutes les langues dans le monde. Le caractère doit être entouré de l' apostrophe ('). Exemple : char cValeur = 'C'; char cValeur = '\x0058'; // Hexa char cValeur = (char)88; // Cast char cValeur = '\u0058'; // Unicode Table ASCII

<u>string</u> (objet de type référence)	Variable	<p>0 à plusieurs caractères. La chaîne de caractères doit être entourée par des guillemets ("). Exemple :</p> <pre>string sTexte = "Bonjour!";</pre> <p>Déclarer une variable de type string sans l'initialiser aura comme valeur null (c.-à-d. rien). Exemple :</p> <pre>string sTexte; // donnera null</pre> <p>Prendre l'habitude de l'initialiser avec la valeur "" (c.-à-d. vide). Exemple :</p> <pre>string sTexte = ""; // donnera vide</pre>
--	----------	--

Initialisation vs Affectation

L'action d'affecter une valeur à une variable **lors de sa déclaration** s'appelle **initialisation d'une variable**.

Exemple d'initialisation :

```
char cLettre = 'A'; //Affectation en même temps que la déclaration.
```

Exemple de **non-initialisation**, mais d'**affectation** :

```
char cLettre;  
cLettre = 'A'; // Affectation après la déclaration (ou plus tard).
```

La portée d'une variable

Selon l'endroit où l'on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien dans une portion confinée de celui-ci (à l'intérieur d'une fonction par exemple), on parle de portée (ou visibilité) d'une variable.

Locale	Globale (ou champ)
Une variable déclarée à l'intérieur d'un bloc d'instructions (dans une fonction ou une boucle par exemple) aura une portée limitée à ce seul bloc d'instructions , c'est-à-dire qu'elle est inutilisable ailleurs, on parle alors de variable locale.	Une variable déclarée au début du code, c'est-à-dire avant tout bloc de donnée, sera globale, on pourra alors l'utiliser à partir de n'importe quel bloc d'instructions. Pour distinguer les variables globales de celles qui sont locales, on ajoute un soulignement devant son nom (<u>_</u>).

Exemple de code en C# :

```
class Program
{
    #region Variables globales (champs)
    static int _iNombre = 0;
    #endregion

    #region Méthodes
    static void DemanderNombre()
    {
        // Demander à l'utilisateur de saisir un nombre.
        do
        {
            Console.WriteLine("Entrer votre nombre: ");
        } while (!int.TryParse(Console.ReadLine(), out _iNombre));
    }
    #endregion
    static void Main(string[] args)
    {
        // Variables locales.
        int iCarré = 0;

        DemanderNombre();

        // Calculer le carré du nombre
        iCarré = _iNombre * _iNombre;

        // Afficher le résultat.
        Console.WriteLine("Le carré du nombre " + _iNombre +
            " est: " + iCarré);
    }
}
```

Zone de déclaration des variables globales (champs).

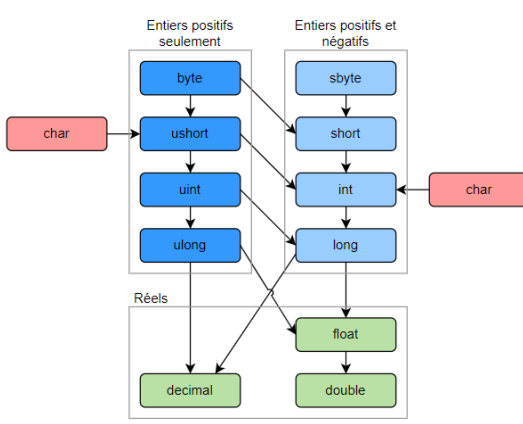
Zone de déclaration des variables locales.

Portée globale

Portée locale

Le transtypage (« casting » en anglais)

Le transtypage est une conversion qui doit être opérée afin de respecter le type d'une variable, ou l'interaction entre plusieurs d'entre elles (lors d'un calcul par exemple).

Implicite	Explicite
<p>Si C# ne détecte aucune contrainte (ex. : perte de données possible), la conversion implicite sera appliquée. Souvent l'implicite passe inaperçue.</p> <p>Exemple :</p> <pre>byte byValeur1 = 100; int iValeur2 = byValeur1;</pre> <p>Implicite</p>  <p>(diagrams.net/draw.io, 2022)</p>	<p>Si C# détecte une contrainte de conversion, le compilateur exigera une conversion explicite (volontaire) en exigeant un opérateur cast « () ». Elle est exigée pour s'assurer que l'usage n'est pas accidentel, car il peut mener à un résultat erroné causé par un dépassement de la capacité du type.</p> <p>Exemple :</p> <pre>int iValeur1 = 100; byte byValeur2 = (byte)iValeur1;</pre> <p>Explicite</p>

Conversion de type *string* vers numérique

Ici, nous parlons de convertir le contenu d'une variable de type *string* vers un type numérique. Il faut utiliser des méthodes de conversion. Exemples :

```
string sValeur = "15"; // Texte
int iValeur = 0;

iValeur = Convert.ToInt16(sValeur); //ou
iValeur = int.Parse(sValeur); // ou
int.TryParse(sValeur, out iValeur); // Retourne vrai si réussi.
```

Génère des exceptions!

Ma recommandation!!!

Les variables de type string

Toute chaîne de caractères doit se retrouver entre guillemets. Il y a des caractères particuliers qui peuvent être utilisés. Ils consistent en une combinaison de deux caractères soit \ (appelé en anglais *Escape* ou *échappement* en français) + un autre caractère. Voici un tableau qui les résume :

Échappement	Caractère produit
\'	Apostrophe (')
\"	Guillemet (")
\\	Barre oblique inversée (\)
\n	Nouvelle ligne
\r	Retour de chariot
\t	Tabulation horizontale

Le mot **concaténation** signifie « coller ensemble deux chaînes de caractères ». En C#, comme dans plusieurs autres langages, cette opération est représentée par l'**opérateur +**. Voici des exemples :

```
string s1 = "Bienvenue au " + "cours de " + "programmation!";
```

Résultat : Bienvenue au cours de programmation!

Accéder à un caractère de la chaîne

Pour accéder à un caractère spécifique d'une chaîne de caractères, il faut utiliser l'**opérateur [] (crochets)**. Exemple : `string sChaine = "caractères";`

	1 ^{er}	2 ^e	3 ^e	4 ^e	5 ^e	6 ^e	7 ^e	8 ^e	9 ^e	10 ^e	11 ^e
	'c'	'a'	'r'	'a'	'c'	't'	'è'	'r'	'e'	's'	'\0'
Index	0	1	2	3	4	5	6	7	8	9	10

Exemple #1 : Obtenir le 5^e caractère de la chaîne « caractères ».

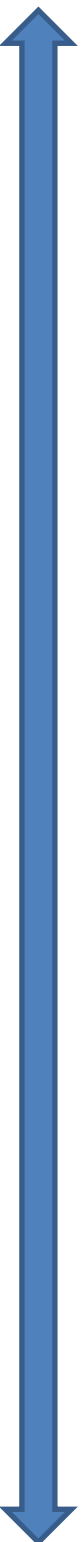
```
string sChaine = "caractères";
char cCar = sChaine[4];
Console.WriteLine(cCar);
```

Résultat : c

Autres : [Empty](#), [Length](#), [ToUpper\(\)](#), [ToLower\(\)](#), [Trim\(\)](#), [IndexOf\(\)](#), [Substring\(\)](#).

Les opérateurs

Voici un rappel des **priorités des opérateurs**. Voir le tableau suivant :

Prior.	Catégorie	Opérateur	Description
	Primaires	x.y	Accès au membre
		f(x)	Méthode et appel de délégué
		a[x]	Tableau et accès d'indexeur
		x++	Post-incrémentation
		x--	Post-décrémentation
		new	Création d'objet et de délégué
		typeof	Obtenir l'objet System.Type pour T
		checked	Évaluer l'expression dans le contexte vérifié (checked)
		unchecked	Évaluer l'expression dans le contexte non vérifié (unchecked)
	Unaires	+x	Identité
		-x	Négation
		!x	Négation logique
		~x	Négation de bits
		++x	Pré-incrémentation
		--x	Pré-décrémentation
		(T)x	Convertir explicitement x en type T
	Multiplication	x * y	Multiplication
		x / y	Division
		x % y	Reste (Modulo)
	Addition	x + y	Addition, concaténation de chaînes, combinaison de délégués
		x - y	Soustraction, suppression de délégué
	Décalage (Shift) – bits	<<	Décalage vers la gauche
		>>	Décalage vers la droite
	Type et relationnel	x < y	Inférieur à
		x > y	Supérieur à
		x <= y	Inférieur ou égal
		x >= y	Supérieur ou égal
		x is T	Retourne la valeur true si x est de type T, faux dans les autres cas
		x as T	Retourne x s'il a le type T ou null si x n'est pas de type T
	Égalité	==	Égal
		!=	Différent de
	Logiques, conditionnelles et nulles	x & y	AND de bits entier, AND logique booléen
		x ^ y	XOR de bits entier, XOR logique booléen
		x y	OR de bits entier, OR booléen logique
		x && y	Évalue y uniquement si x est vrai
		x y	Évalue y uniquement si x est faux
		x ?? y	Évalue y si x est null, sinon évalue x
		x ? y : z	Évalue à y si x est vrai, z si x est faux
	Assignation ou Affectation	x = y	Assignation
		x op= y	Assignation composée. Prend en charge ces opérateurs : +=, -=, *=, /=, %=, &=, =, !=, <=, >=
Basse			

Structures alternatives (Conditions)

Type	Description	Exemple(s) en C#
Simple SI ... ALORS ... FIN SI	<p>La ou les instructions seront exécutées si le résultat de l'expression booléenne est vraie (<i>true</i>).</p> <pre> if (expression) Instruction; if (expression) { Instruction 1; Instruction 1; ... } </pre>	<pre> int iNb = 20; if (iNb == 20) Console.WriteLine("Nombre: 20"); int iNb = 20; if (iNb == 20) { Console.WriteLine("Nombre: 20"); iNb = 0; Console.WriteLine("Remis à 0."); } </pre>
Complexe SI ... ALORS ... SINON ... FIN SI	<p>La ou les instructions contenues dans le bloc 1 seront exécutées si le résultat de l'expression booléenne est vraie (<i>true</i>) sinon les instructions du bloc 2 seront exécutées.</p> <pre> if (expression) Instruction 1; else Instruction 2; if (expression) { Instruction 1a; Instruction 1b; } else { Instruction 2a; Instruction 2b; } </pre>	<pre> byte byVitesse = 30; if (byVitesse <= 30) // Zone scolaire Console.WriteLine("Vitesse OK."); else Console.WriteLine("Dépassée!!!"); float fPrixTicket = 0; // Amende if (byVitesse <= 30) // Zone scolaire { Console.WriteLine("Vitesse OK."); fPrixTicket = 0; } else { Console.WriteLine("Dépassée"); fPrixTicket = 50; } </pre>
Imbriquée (plusieurs branches)	<p>Il y a structure alternative imbriquée lorsque les branches d'une alternative font appel à une décision. Dès qu'il y a présence d'un SI à l'intérieur d'un autre SI (que ce soit après le ALORS, ou le SINON, ou les deux), la structure est dite « imbriquée ». Exemple :</p> <pre> byte byVitesse = 30; if (byVitesse <= 30) // Zone scolaire Console.WriteLine("Vitesse OK."); else { if (byVitesse >= 50) Console.WriteLine("Trop vite!"); else Console.WriteLine("Élevée"); } if (byAge <= 18) { if (byAge <= 12) Console.WriteLine("Enfant"); else Console.WriteLine("Ado"); } else { if (byAge <= 65) Console.WriteLine("Adulte"); else Console.WriteLine("Âge d'or"); } </pre>	<pre> byte byVitesse = 30; if (byVitesse <= 30) Console.WriteLine("Vitesse OK."); else if (byVitesse >= 50) Console.WriteLine("Très élevée"); else Console.WriteLine("Élevée"); if (byAge <= 12) Console.WriteLine("Enfant"); else if (byAge <= 18) Console.WriteLine("Ado"); else if (byAge <= 65) Console.WriteLine("Adulte"); else Console.WriteLine("Âge d'or"); </pre>

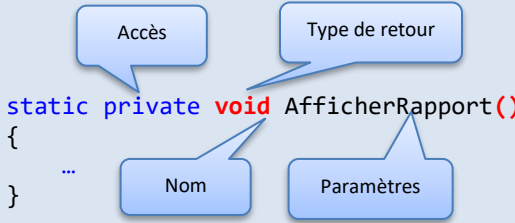
Multiple	<p>La structure décisionnelle multiple est une spécialisation de l'instruction SINON SI. Elle offre la possibilité d'exécuter différents blocs d'instructions en fonction d'une valeur spécifiée dans une variable. Elle a un rôle d'aiguilleur lorsqu'une condition offre plusieurs voies de sortie.</p> <p>En C# :</p> <pre> switch (variable) { case Valeur1: // Instruction(s) #1 break; case Valeur2: // Instruction(s) #2 break; case Valeur3: // Instruction(s) #3 break; case Valeur4: // Instruction(s) #4 break; default: // Instruction(s) - Autrement break; } </pre>
-----------------	--

Opérateurs logiques conditionnels

Opérateur	Description															
ET (&&)	<p>L'opérateur logique conditionnel « ET » placé entre deux conditions donnera une réponse VRAIE si et seulement si les <u>deux conditions</u> retournent VRAI. Si une seule condition retourne FAUX entre les deux, alors ce sera FAUX. Voici la table :</p> <table><tr><th>A</th><th>B</th><th>Résultat (A && B)</th></tr><tr><td>false</td><td>false</td><td>false</td></tr><tr><td>false</td><td>true</td><td>false</td></tr><tr><td>true</td><td>false</td><td>false</td></tr><tr><td>true</td><td>true</td><td>true</td></tr></table>	A	B	Résultat (A && B)	false	false	false	false	true	false	true	false	false	true	true	true
A	B	Résultat (A && B)														
false	false	false														
false	true	false														
true	false	false														
true	true	true														
OU ()	<p>L'opérateur logique conditionnel « OU » placé entre deux conditions donnera une réponse VRAIE <u>si au moins l'une des deux conditions est VRAIE</u>. Voici la table :</p> <table><tr><th>A</th><th>B</th><th>Résultat (A B)</th></tr><tr><td>false</td><td>false</td><td>false</td></tr><tr><td>false</td><td>true</td><td>true</td></tr><tr><td>true</td><td>false</td><td>true</td></tr><tr><td>true</td><td>true</td><td>true</td></tr></table>	A	B	Résultat (A B)	false	false	false	false	true	true	true	false	true	true	true	true
A	B	Résultat (A B)														
false	false	false														
false	true	true														
true	false	true														
true	true	true														
NON (!)	<p>L'opérateur logique conditionnel « NON » devant une condition donnera une réponse VRAIE si la condition est FAUSSE et donnera FAUX si la condition est VRAIE (donne l'opposé ou le complément). Voici la table de vérité :</p> <table><tr><th>A</th><th>Résultat (!A)</th></tr><tr><td>false</td><td>true</td></tr><tr><td>true</td><td>false</td></tr></table>	A	Résultat (!A)	false	true	true	false									
A	Résultat (!A)															
false	true															
true	false															

Méthodes et fonctions

Une méthode **est un bloc de code contenant une série d'instructions**. Elle définit les actions ou les opérations supportées par la classe. Une méthode se définit en spécifiant **son niveau d'accès**, **son type retourné** (si aucun, utiliser *void*), **son nom**, sa **liste de paramètres** entre parenthèses (si aucun, inscrire ()) et ses **instructions entre accolades**.

Type	Description	Exemple C#
Sans retour et sans paramètre		 <pre>static private void AfficherRapport() { ... }</pre>
Sans retour et avec paramètres	Pour ajouter de la flexibilité à la conception d'une classe, les méthodes peuvent contenir des paramètres facilitant le passage d'informations. Pour se faire ils doivent être déclarés entre les () de la méthode. Il est conseillé de ne pas dépasser plus de 5 paramètres pour une méthode . Parfois si une méthode nécessite plus de 5 paramètres, il faut alors considérer d'autres avenues pour passer l'information (ex. : Se créer une classe, une structure, un tableau, etc.). Lors de la déclaration, chaque paramètre est séparé par une virgule .	<pre>static private void AfficherRapport(bool bFichier) { if (bFichier) { // Codes pour écrire dans un fichier } else { // Codes pour afficher à l'écran } }</pre>
Avec retour (sans ou avec paramètres) FONCTION!!!	Une méthode qui retourne une valeur doit contenir l'instruction return suivi de la valeur ou variable qui contient la valeur à retourner. Une fois cette instruction atteinte, le programme sort de cette méthode.	<pre>static private int Additionner(int iNb1, int iNb2) { return iNb1 + iNb2; } ... int iTotal = Additionner(3, 7); // L'appel. Va donner 10.</pre>

Notions de compteur

Un compteur est un dispositif, registre ou partie de mémoire qui permet de connaître, à un moment donné, le nombre d'occurrences d'un signal ou d'un événement donné. Un compteur emmagasine un nombre de départ (ex. : une variable) qui est augmenté (ici, on dit « **Incrémenter** ») ou bien diminué (ici on dit « **Décrémenter** ») selon un intervalle régulier.

Incrémentation	Décrémentation
En C# (optimal): <pre>int iNb = 10; iNb++; // augmente de 1. iNb+=5; // augmente de 5.</pre>	En C# (optimal): <pre>int iNb = 10; iNb--; // diminue de 1. iNb-=5; // diminue de 5.</pre>

Structures répétitives (Boucles)

Type	Description	Exemple(s)
TANT QUE (while)	<p>La particularité de cette boucle est que l'évaluation de la condition se fait au début.</p> <p>On utilise une boucle lorsque le nombre d'itérations est indéterminé.</p> <pre>while (expression) { // DÉBUT du bloc. // Instruction(s) // ... }</pre>	<pre>// Accumuler les positifs int iSomme = 0; int iNombre = 0; while (iNombre >= 0) { iSomme += iNombre; Console.Write("Nombre? "); iNombre = int.Parse(Console.ReadLine()); } Console.WriteLine("Somme: " + iSomme);</pre>
JUSQU'À (do...while)	<p>La particularité de cette boucle est que l'évaluation de la condition se fait à fin. Donc la séquence d'instructions sera exécutée au moins une fois dans la boucle.</p> <pre>do { // DÉBUT du bloc. // Instruction(s) } // FIN du bloc. while (expression);</pre>	<pre>// Accumuler les positifs int iSomme = 0; int iNombre = 0; do { iSomme += iNombre; Console.Write("Nombre? "); iNombre = int.Parse(Console.ReadLine()); } while (iNombre >= 0); Console.WriteLine("Somme: " + iSomme);</pre>

POUR (for)	<p>Ce type de boucle est fortement associé aux notions de compteur, car elle implémente un compteur afin de viser et s'arrêter à une valeur bien précise. Elle permet d'effectuer un nombre déterminé d'itérations.</p> <pre> for (cpt = début; cpt < fin; cpt = cpt + PAS) { // Instruction(s) 1 } </pre>	<pre> // Afficher un nombre de 1 à 5. for (int iNb = 1; iNb <= 5; iNb++) { // Afficher la valeur. Console.WriteLine(iNb); } // Afficher un nombre de 15 à 7. for (int iNb = 15; iNb >= 7; iNb--) { // Afficher la valeur. Console.WriteLine(iNb); } // Afficher les nombres impairs // de 1 à 13. for (int iNb = 1; iNb <= 13; iNb+=2) { // Afficher la valeur. Console.WriteLine(iNb); } </pre>
---------------	--	---

Les fichiers de type texte

L'utilisation de fichiers texte demeure une façon simple et facile d'alimenter un programme pour lui faire faire divers traitements. Les fichiers texte nous permettent d'y déposer des données, évitant ainsi d'avoir à les resaisir à nouveau. Dans cette section, nous allons voir les objets qui permettent de lire le contenu d'un fichier texte et d'écrire du contenu dans un fichier.

namespace à utiliser

Pour avoir accès aux classes qui permettent la manipulation de fichier, il faut inclure la clause suivante dans l'entête du module:

```
using System.IO; // Donne accès aux fichiers
```

Les classes **File** et **Directory**

Ces deux classes possèdent plusieurs méthodes favorisant la gestion et la manipulation de fichiers.

Voici les méthodes les plus utilisées :

Méthode (statique)	Description
File.Exists(string)	En passant le nom du fichier (avec son répertoire) en paramètre, cette méthode permet d'indiquer si le fichier existe ou non sur le disque. Exemple : <pre>if (!File.Exists("test.txt")) { Console.WriteLine("Le fichier n'existe pas!"); return; // Permet de quitter la méthode. }</pre>
Directory.Exists(string)	Cette méthode permet de vérifier l'existence du répertoire passé en paramètre. Exemple : <pre>if (!Directory.Exists("c:\\temp")) { Console.WriteLine("Le répertoire n'existe pas!"); return; // Permet de quitter la méthode. }</pre>
Directory.GetFiles(string)	Retourne un tableau de type string contenant la liste des noms de fichiers contenus dans le répertoire spécifié. Exemple : <pre>string[] asNoms = Directory.GetFiles("C:\\temp");</pre> <p>Si le répertoire contient les fichiers: auto.txt, bateau.pdf et avion.docx. Alors l'information contenue dans le tableau sera :</p> <pre>asNoms[0] < "C:\\temp\\auto.txt" asNoms[1] < "C:\\temp\\bateau.pdf" asNoms[2] < "C:\\temp\\avion.docx"</pre>

La classe StreamReader

Utiliser [StreamReader](#) pour lire des lignes d'informations à partir d'un fichier texte standard. Sauf spécification contraire, l'encodage par défaut de StreamReader est [UTF-8](#), plutôt que la page de codes ANSI par défaut du système actuel. UTF-8 gère correctement les caractères Unicode et fournit des résultats cohérents sur les versions localisées du système d'exploitation. Cet objet possède plusieurs constructeurs et méthodes qui sont détaillés sur MSDN. Voici les plus couramment utilisés dans l'utilisation de fichier de type TEXTE:

Constructeur	Description
StreamReader(string)	Il existe plusieurs constructeurs différents concernant cet objet, mais celui qui est couramment utilisé, pour les fichiers de type Texte, est celui-ci. Il suffit de lui passer en paramètre le <u>nom du fichier à ouvrir</u> lors de l'instanciation. Exemple : <pre>StreamReader fichier = new StreamReader("test.txt");</pre>

Méthode	Description
Read()	Cette méthode sert à lire <u>caractère par caractère</u> le contenu du fichier texte. S'il n'y a plus de caractères à lire, la méthode retourne -1 .
ReadLine()	Cette méthode sert à lire <u>une ligne complète</u> (c.-à-d. une ligne est définie comme une séquence de caractères suivie d'un saut de ligne ("\n"), d'un retour chariot ("\r"), ou d'un retour chariot immédiatement suivi d'un saut de ligne ("\r\n").) contenu dans le fichier texte. S'il n'y a plus de lignes à lire, la méthode retourne null .
ReadToEnd()	Cette méthode effectue la lecture du début jusqu'à la fin du fichier texte. Retourne donc l'ensemble du contenu. Si la position actuelle est à la fin du flux, retourne une chaîne vide.
Close()	Cette méthode permet de fermer le fichier en cours. Elle effectue un Dispose() implicite. Et vice versa. Si vous appelez un Dispose() sans avoir appelé un Close() avant, Dispose() appellera un Close() implicite.

Propriété	Description
EndOfStream	Cette propriété permet de savoir si la fin du fichier a été atteinte ou non. Souvent utiliser pour arrêter la lecture du fichier.

Exemple #1 : Lire un fichier pour en afficher le contenu dans une boîte de texte.

```
// Ouverture du fichier
StreamReader fichierEntrant = new StreamReader("test.txt");
string sLigne = "";

// Tant que la fin du fichier n'est pas atteinte, on lit.
while (!fichierEntrant.EndOfStream)
{
    sLigne = fichierEntrant.ReadLine(); // Lire chaque ligne
    Console.WriteLine(sLigne + "\r\n"); // Afficher la ligne
}

fichierEntrant.Close();
```

La classe StreamWriter

[StreamWriter](#) est conçu pour la sortie de caractères dans un encodage particulier. Cet objet possède plusieurs constructeurs et méthodes qui sont détaillés sur MSDN. Voici les plus couramment utilisés dans l'utilisation de fichier de type TEXTE:

Constructeur	Description
<u>StreamWriter(string)</u>	Permet d'instancier l'objet en utilisant le nom de fichier passé en paramètre et en l'ouvrant en mode écriture. Si le fichier existe, il sera écrasé. Sinon un nouveau fichier est créé. Exemple : <code>StreamWriter fichier = new StreamWriter("sortie.txt");</code>
<u>StreamWriter(string, bool)</u>	Permet d'instancier l'objet en utilisant le nom de fichier passé en paramètre et en l'ouvrant en mode Ajout (<i>Append</i>) si le deuxième paramètre est <i>true</i> . Si le fichier existe, le pointeur sera placé à la fin du fichier pour ajouter, sinon le fichier est créé. Exemple : <code>StreamWriter fichier = new StreamWriter("sortie.txt", true);</code>

Méthode	Description
Write()	Écrit dans le fichier à la suite du texte déjà existant.
WriteLine()	Écrit dans le fichier la chaîne de texte suivi d'un saut de ligne et d'un retour de chariot (" \r\n ").
Flush()	Toute écriture vers un fichier se retrouve dans une zone mémoire nommée « tampon » en attendant que le système d'exploitation décide de l'écrire dans le fichier et vider cette mémoire. Cependant, nous pouvons forcer le SE à écrire dans le fichier au moment qu'il nous convient en utilisant cette méthode après un Write ou WriteLine .
Close()	Cette méthode permet de fermer le fichier en cours. Elle effectue un Dispose() implicite. Et vice versa. Si vous appelez un Dispose() sans avoir appelé un Close() avant, Dispose() appellera un Close() implicite.

Exemple #1 : Écrire 5 phrases dans un fichier texte.

```
const int NB_FOIS = 5; // Indique le nombre de saisies

StreamWriter fichierSortant = new StreamWriter("test_sortie.txt");

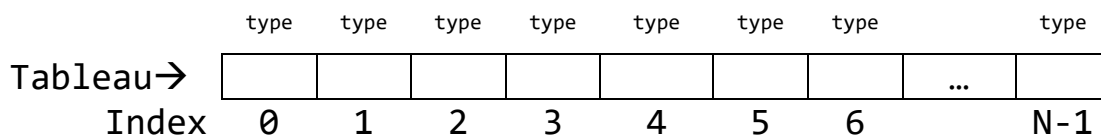
string sPhrase = "";
for (int iNb = 0; iNb < NB_FOIS; iNb++)
{
    Console.WriteLine("Entrer une phrase : ");
    sPhrase = Console.ReadLine();
    fichierSortant.WriteLine(sPhrase);    // Écrire la phrase
    fichierSortant.Flush(); // Obliger le SE à écrire maintenant.
}
fichierSortant.Close();
```

Les tableaux

Définition d'un tableau 1D

Un tableau *est un ensemble ordonné qui contient un nombre fixe d'éléments de même type*. Un tableau à une dimension est couramment appelé : **Vecteur**. Le terme anglais utilisé pour ce type de structure est **array**. Quelques règles :

- Les valeurs contenues dans le tableau seront de même type.
- Les tableaux sont indexés sur un intervalle fixe à partir de zéro.
- l'**opérateur** associé aux tableaux est les **crochets []**.
- longueur N, l'index associé varie dans l'intervalle [0 à N – 1].
- Le mot clef **new** correspond à la création d'un nouvel objet.
- Selon la notation Hongroise, le préfixe **a** ou **ar** pour **array**



Exemple #1 : Créer un tableau qui contiendra 3 notes d'un étudiant.

Indique que la variable sera un ta-

Crée le tableau en mémoire

Définit la dimension de ce tableau.

```
byte[] abyNote = new byte[3];
```

En mémoire :

<pre> abyNote[0] = 87; // Note 1 abyNote[1] = 93; // Note 2 abyNote[2] = 100; // Note 3 </pre>	<p>abyNote →</p> <table border="1" style="display: inline-table;"> <tr> <td></td><td></td><td></td> </tr> <tr> <td>87</td><td>93</td><td>100</td> </tr> <tr> <td>Index</td><td>0</td><td>1</td><td>2</td> </tr> </table>				87	93	100	Index	0	1	2
87	93	100									
Index	0	1	2								

byte byte byte

On peut obtenir la dimension du tableau en utilisant la propriété **Length**.

Exemple :

```
int iLongueur = abyNote.Length; // Ici on obtient 3.
```

Lors de la déclaration d'un vecteur, nous avons la possibilité d'initialiser le tableau avec des valeurs en énumérant les valeurs entre accolades {} et séparées par des virgules. Voici des exemples :

```
byte[] abyNote = { 87, 93, 100 };
string[] asPrénom = { "Lise", "Marc", "Luc", "Paul", "Lina" };
```

Tableau à deux dimensions (2D)

Dans cette séquence, nous allons traiter du tableau à deux dimensions. Un tableau à deux dimensions est couramment appelé : **Matrice** (peut avoir plus que 2 dimensions)

Le tableau à deux dimensions, ou matrice, structure l'information en mémoire de la façon suivante :

	Index	0	1	2	3	4	5	...	Colonne N-1
Matrice → (type)	0							...	
	1							...	
	2							...	
	3							...	
	4							...	
		
	M-1								

Ligne

Déclaration :

```
type[,] aMatrice = new type[M, N];
```

Exemple #1 : Créer un tableau qui contiendra 3 notes pour 4 étudiants.

Indique que la variable sera un ta-

Crée le tableau en mémoire

Définit les dimensions: 4 lignes et 3 colonnes.

```
byte[,] abyNote = new byte[4,3];
```

En mémoire :

```

abyNote[0,0] = 87; // Étudiant 1 - Note 1
abyNote[0,1] = 93; // Étudiant 1 - Note 2
abyNote[0,2] = 100; // Étudiant 1 - Note 3
abyNote[1,0] = 65; // Étudiant 2 - Note 1
abyNote[1,1] = 60; // Étudiant 2 - Note 2
abyNote[1,2] = 72; // Étudiant 2 - Note 3
abyNote[2,0] = 58; // Étudiant 3 - Note 1
abyNote[2,1] = 91; // Étudiant 3 - Note 2
abyNote[2,2] = 84; // Étudiant 3 - Note 3
abyNote[3,0] = 75; // Étudiant 4 - Note 1
abyNote[3,1] = 94; // Étudiant 4 - Note 2
abyNote[3,2] = 98; // Étudiant 4 - Note 3

```

	Index	0	1	2
abyNote → (byte)	0	87	93	100
	1	65	60	72
	2	58	91	84
	3	75	94	98

Méthode GetLength

Pour obtenir la longueur d'une dimension, il faut utiliser **GetLength(dimension)** en lui passant la position de la dimension. La première dimension est représentée par 0, la deuxième par 1, etc. Exemple :

```
byte byNbLignes = (byte)abyNote.GetLength(0); // ici va retourner 4.
byte byNbColonnes = (byte)abyNote.GetLength(1); // ici va retourner 3.
```

Les index

Par la suite, les éléments du tableau peuvent être accédés par l'indexeur. Exemple: Calculer la moyenne de chaque étudiant.

```
byte byNbLignes = (byte)abyNote.GetLength(0 /*1ère dimension*/); // ici va retourner 4.
byte byNbColonnes = (byte)abyNote.GetLength(1 /*2e dimension*/); // ici va retourner 3.

// Boucler sur toutes les lignes
for (byte byPosLigne = 0; byPosLigne < byNbLignes; byPosLigne++)
{
    float fMoyenne = 0.0f; // On réinitialise pour chaque étudiant.

    // Boucler sur les notes de l'étudiant
    for (byte byPosColonne = 0; byPosColonne < byNbColonnes; byPosColonne++)
        fMoyenne += abyNote[byPosLigne, byPosColonne];
    Console.WriteLine("Moyenne de l'étudiant #" + (byPosLigne + 1) + ": " +
        (fMoyenne / (float)byNbColonnes).ToString("0.0"));
}
```

La liste générique

La classe « [List](#) » (MSDN, 2013) représente une liste fortement typée accessible via un index. Contrairement à un tableau, cette liste peut contenir un nombre indéfini d'éléments. **Sa taille augmente de façon dynamique.**

Espace de nom

Pour simplifier la syntaxe rattachée à l'utilisation de la classe List, vous pouvez inscrire la ligne de code suivante dans l'entête du programme :

```
using System.Collections.Generic;
```

Déclaration d'une liste

Voici la syntaxe d'une déclaration d'une liste :

```
List<T> variable = new List<T>();
```

T représente le type de données qui peut être contenu dans la liste. Les données seront toutes de même type. Exemples :

```
List<string> animaux = new List<string>();
List<int> nombres = new List<int>();
List<float> montants = new List<float>();
```

Propriétés d'une liste

Nom	Description
Count	Elle est de type <i>int</i> . Permet d'obtenir le nombre d'éléments réellement contenus dans List<T>. Exemple : <pre>List<string> animaux = new List<string>(); int iNbObjets = animaux.Count; // Ici va donner 0.</pre>

Quelques méthodes de base

Voici maintenant quelques méthodes de base qui vous permettront de manipuler facilement cette liste générique.

Nom	Description
Add	Permet d'ajouter un objet à la fin de la liste. Exemple : <pre>List<string> animaux = new List<string>(); animaux.Add("Chien"); animaux.Add("Chat"); animaux.Add("Cheval"); animaux.Add("Oiseau");</pre>
Clear	Permet de supprimer tout le contenu d'une liste. Exemple : <pre>animaux.Clear(); // Liste se vide.</pre>
Contains	Permet d'indiquer la présence ou non de l'objet recherché dans la liste. Retourne Vrai si présent, sinon Faux. Exemple : <pre>if (animaux.Contains("Souris")) Console.WriteLine("OUI"); else Console.WriteLine("NON");</pre>
IndexOf	Permet de chercher un objet dans la liste et de retourner la position de la première occurrence (son index à base 0). Exemple : <pre>int iIndex = animaux.IndexOf("Cheval"); // retourne 2 (3^e pos)</pre>
Insert	Permet d'insérer un nouvel élément dans la liste à l'index spécifié. Exemple : <pre>animaux.Insert(1, "Chameau");</pre>
Remove	Permet de retirer la première occurrence d'un objet de la liste. Retourne Vrai si la suppression est réussie, sinon Faux. Exemple : <pre>if (animaux.Remove("Chat"))</pre>

	<pre>Console.WriteLine("Retiré!"); else Console.WriteLine("Non trouvé!");</pre>
<u>RemoveAt</u>	<p>Permet de supprimer l'objet se trouvant directement à l'index spécifié. Exemple :</p> <pre>animaux.RemoveAt(1); // 2e élément retiré</pre>

Exemple #1 : Ajouter 5 couleurs dans une liste et afficher son contenu.

```
List<string> couleurs = new List<string>(); // Déclaration de la liste.
```

```
// Ajouter les mots
```

```
couleurs.Add("Rouge");
```

```
couleurs.Add("Bleu");
```

```
couleurs.Add("Jaune");
```

```
couleurs.Add("Vert");
```

```
couleurs.Add("Blanc");
```

```
// Afficher le contenu de la liste.
```

```
foreach (string sCouleur in couleurs)
```

```
    Console.WriteLine(sCouleur + "\n");
```

```
//ou
```

```
for (int iIndex = 0; iIndex < couleurs.Count; iIndex++)
```

```
    Console.WriteLine(couleurs[iIndex] + "\n");
```

Exercices

Exercice 1. Créer un programme qui consiste à demander à l'utilisateur de saisir un prix unitaire, une quantité et d'afficher le sous-total avant taxes, montant de la TVQ, montant de la TPS et le total.

Buts : La séquence, initiation entrée et sortie, variables locales, calculs simples.

Exercice 2. Créer un programme qui consiste à afficher le menu et attendre le choix de l'utilisateur. Voici le menu :

```
==== Menu ====  
1) Additionner  
2) Soustraire  
3) Multiplier  
4) Diviser  
9) Quitter  
Votre choix : _
```

Pour les 4 premières opérations, vous devrez demander de saisir Nombre1 et Nombre2 et faire l'opération. Ensuite, afficher le résultat (faire une pause pour que l'utilisateur puisse avoir le temps de voir la réponse). Ensuite, réafficher le menu en vidant l'écran. Lorsque l'utilisateur saisi 9, le programme doit quitter. **Utiliser obligatoirement une structure décisionnelle multiple.** Si l'utilisateur n'entre pas une bonne valeur, afficher un message d'erreur et revenir au menu.

Créer une méthode **AfficherMenu** (servant à contenir le code d'affichage du menu) et qui sera appelée dans le programme principal (Main).

Buts : Boucle, vider l'écran et une méthode simple, structure décisionnelle multiple (switch).

Exercice 3. Créer un programme portant le nom de l'exercice. Le programme consiste à demander à l'utilisateur de saisir 7 (définir une constante et l'utiliser) prénoms (un prénom par ligne) et de les ajouter dans une liste. Par la suite, afficher chaque prénom à l'envers (boucle obligatoire) à partir de cette liste.


Buts : Boucles, Utiliser l'objet **List**, constante

Exercice 4. Créer un programme qui consiste à remplir un tableau 1D à partir d'une chaîne de caractères saisie par l'utilisateur. Exemple : 45, 67, 34, 40, 61, 78, 99. Ce tableau doit être déclaré comme champ (global) obligatoirement. Par la suite, vous devez déclarer une méthode dont la signature est :

```
void Permuter(int iIndexSource, int IndexDest);
```

Cette méthode devra prendre la valeur à l'index Source pour la déposer à l'index Destination et vice versa. Exemple avec Source=2 et Destination = 5:

0	1	2	3	4	5	6
45	67	34	40	61	78	99



Après permutation : 45 67 78 40 61 34 99

Fonctionnements demandés :

- 1) 1^{re} fois demander les nombres sur une ligne et remplir le tableau (variable globale exigée).
- 2) Dans une boucle :
 - a. Afficher le tableau (effacer l'écran à chaque fois)
 - b. Demander l'index Source
 - c. Demander l'index Destination
 - d. Faire appel à la méthode Permuter
 - e. Revenir au point A. Un négatif dans la source et/ou destination fait quitter le programme.

Buts : Tableau 1D, méthode avec paramètres, champ.

Exercice 5. Créer un programme qui consiste à lire le contenu d'un fichier texte et insérer les valeurs dans un tableau 2D (int). Le programme doit détecter automatique le nombre de lignes et de colonnes. Par la suite, vous devez déclarer une fonction dont la signature est :

```
int[,] Retourner(int[,] aiTableauSource);
```

Cette fonction doit prendre un tableau en paramètre et le retourner. Exemple :

Source (2 X 4):

5	6	16	43
4	0	11	26

Résultat (4 X 2):

5	4
6	0
16	11
43	26

Créer un fichier de type texte avec les valeurs indiquées par la Source (ici haut). Les valeurs sont séparées par une espace.

Affcher le tableau Source et le tableau Destination. Après l'affichage faire une pause et quitter le programme.

Buts : Tableau 2D, fonction avec paramètre, fichier entrant

Bibliographie

diagrams.net/draw.io. (2022, 01 05). *Usage terms for diagrams created in diagrams.net*.

Récupéré sur diagrams.net: <https://www.diagrams.net/doc/faq/usage-terms>

MSDN. (2013, 01 01). *List<T>, classe*. Consulté le 02 17, 2013, sur MSDN:

[http://msdn.microsoft.com/fr-fr/library/vstudio/6sh2ey19\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/vstudio/6sh2ey19(v=vs.110).aspx)

Wikipédia. (s.d.). *Variable globale*. Consulté le 12 20, 2011, sur Wikipédia:

http://fr.wikipedia.org/wiki/Variable_globale