

Haute Ecole de la Ville de Liège

Catégorie Technique

La programmation orientée objet par l'intermédiaire de C#

A l'usage des étudiants en 2^{ème} Informatique et Systèmes (Technologies de l'Informatique)



Année académique 2017-2018
Patrick Alexandre

0. Introduction

0.1. Historique

Avant de passer à la programmation à proprement parler, rappelons les quelques grandes lignes de l'histoire de la programmation orientée objet.

L'approche objet est une idée qui a fait ses preuves. Simula a été le premier langage de programmation à implémenter le concept de classes et est apparu dès 1967.

En 1976, le développement informatique, version objet, connaît une seconde jeunesse avec l'apparition de Smalltalk qui implémente l'encapsulation, l'agrégation, et l'héritage, notions essentielles dans l'approche orientée objet sur lesquels nous allons revenir. Signalons également que d'autres langages orientés objet ont été mis au point dans le cadre de recherches universitaires (Eiffel, Objective C, Loops, ...).

L'aventure du C++¹ commença en 1983 sous la houlette de Bjarne Stroustrup, des laboratoires AT&T Bell. L'objectif de départ est de se reposer complètement sur le langage C et lui adjoindre des classes analogues à celles présentes dans le langage Simula. Comme le C, il a fait l'objet d'une standardisation. Il est donc compatible et, par conséquent, est portable au niveau du source, l'exécutable dépendant fortement du système d'exploitation.

Début des années 1990, le langage Java fait son apparition sous la houlette de James Gosling, de Sun. Portant le doux nom de Oak à son origine, il a été conçu répondre aux besoins de l'électronique qui commençait à être friande d'applications embarquées. Java est devenu lui-même avec l'essor d'Internet. En ce qui concerne les habitudes de programmation, Java se différencie de ses illustres prédécesseurs par l'intégration, en standard, d'un grand nombre de ressources (classes et méthodes). De plus, la portabilité est améliorée en ce sens qu'une machine virtuelle prend en charge la conversion du code source pré-compilé vers le langage machine propre à la plate-forme.

Vient la dernière partie qui nous concerne : C#. En octobre 1996, Microsoft débauche, notamment, Anders Hejlsberg de chez Borland où des produits de RAD (Rapid Application Development) avaient été élaborés sous sa direction (Delphi, C++ Builder). Le but du jeu est de mettre au point développer une plate-forme de développement performante car Visual Studio était à la traîne dans ce créneau. C'est ainsi que, en juin 2000, Microsoft annonça l'architecture .NET et le langage C#.

¹ Pour information, le nom C++ a été déterminé par Rick Mascitti pour faire référence au langage C amélioré de 1, d'où la présence de l'opérateur d'incrémentation bien connu des développeurs en C. A l'origine, ce langage est destiné au système d'exploitation Unix. Les versions 1.1 et 1.2 du langage C++ voient le jour, respectivement, en 1986 et 1987 tandis que la version 2.0 est disponible en 1989.

En ce qui nous concerne, le langage C# est largement inspiré de Java même si la lignée officielle² est $C \rightarrow C++ \rightarrow C\#$. Il nécessite une machine virtuelle (Just In Time Compiler) et est accompagné d'un très grand nombre de ressources.

0.2. Contexte

Ces notes de cours sont destinées aux étudiants de 2^{ème} Informatique et Systèmes, orientation Technologies de l'Informatique. La première année, en programmation, leur fait découvrir le langage C. Jusqu'à l'année académique 2003-2004, le langage C++³ était le langage de programmation de 2^{ème} année.

Les raisons de ces choix étaient multiples :

- la syntaxe (étude des règles qui président à l'ordre des mots et à la construction d'une phrase dans une langue) et la sémantique (étude de la langue du point de vue du sens) sont les mêmes pour C et C++,
- tout programme rédigé en C peut être incorporé dans une nouvelle application écrite en C++,
- un large champ d'action depuis les applications scientifiques jusqu'aux applications de gestion en passant par le pilote de matériel, les O.S. modernes et les logiciels de jeux..

L'année 2004-2005 vit apparaître, en 2^{ème} année, le langage C#. Cette introduction se justifiait par

- la montée en puissance de C# au niveau de la vie professionnelle comme en témoignent les offres d'emploi et la présence de plus en plus importante d'articles le concernant sur Internet,
- les cours de développement d'applications dédiées à Internet⁴ de 3^{ème} année qui s'appuient sur l'architecture .NET et, plus particulièrement sur ASP .NET et ADO .NET.

Les étudiants se trouvant confrontés à l'apprentissage de 3 langages fort semblables, l'équipe enseignante a décidé d'abandonner le langage C++ dès 2005-2006. Il convient cependant de noter que les raisons du choix de C# sont similaires à celles présentées ci-dessus pour C++.

Pour terminer avec cette présentation du contexte, signalons également que ces notes de cours s'appuient sur la connaissance du langage C qu'ont nos étudiants. Cela n'empêche pas de décrire les instructions de base du langage mais permet d'aller, à certains moment, un peu plus vite. Nous pensons notamment au chapitre 2 qui s'appuie sur certains exemples parfaitement compréhensible, dans les grandes lignes, pour quelqu'un qui ne connaît pas C# mais bien C.

2 Histoire d'entretenir la polémique entre libre et propriétaire, C#est présenté par Microsoft comme le *Java killer* ... L'histoire nous dira qui de Bill ou de Scott gagnera la guerre des langages mais il est dommage qu'il soit plus facile de faire la guerre que la paix ...

3 Il convient d'ajouter JavaScript mais dans une moindre mesure.

4 La partie Open Source est, quand à elle, assurée par le couple (PHP, MySql).

0.3. Présentation

C# est un langage de programmation. Par conséquent, il permet de traduire des instructions exprimées dans un langage compréhensible par l'être humain en instructions devant être réalisées par l'ordinateur. Le programmeur, pour mener cette tâche à bien, doit rédiger son code source en suivant (scrupuleusement) un certain nombre de règles. Cet ensemble de commandes est enregistré dans des fichiers texte en ASCII pur et portent habituellement l'extension cs.

Ce langage de programmation est, en plus, un langage compilé. Cela signifie que tout programme est soumis au compilateur qui analyse le code pour déceler les éventuelles erreurs syntaxiques. Une fois le programme correct du point de vue de la syntaxe, une première traduction est effectuée selon un fichier compilé. Avant les machines virtuelles, un exécutable (programma en langage machine) était généré. C# et Java soumettent un fichier pré-compilé à la machine virtuelle qui effectue la traduction en langage machine.

Comme tout langage qui se respecte, C# est caractérisé⁵ par

- **l'*exactitude***
aptitude à fournir des résultats voulus dans des conditions normales d'utilisation (données conformes aux spécifications),
- **la *robustesse***
aptitude à réagir convenablement si on s'écarte des conditions normales d'utilisation,
- **la *fiabilité***
l'ensemble du programme ne peut se retrouver inutilisable si une partie de l'application est prise en défaut de fonctionnement,
- **la *maintenabilité***
les corrections et les modifications des programmes en cours d'utilisation sont un point essentiel pour des réalisations professionnelles et il convient de ne pas mettre en danger l'ensemble de l'application lors de telles opérations,
- **l'*extensibilité***
faculté de pouvoir adjoindre des données et/ou des fonctionnalités supplémentaires sans risquer de compromettre le fonctionnement de l'ensemble,
- **la *réutilisabilité***
possibilité de pouvoir utiliser des développements réalisés auparavant pour une nouvelle application, dans un autre contexte,
- **la *portabilité***
possibilité d'utiliser le même programme sur des machines et/ou des environnements différents,
- **l'*efficience***
la possibilité d'exploiter au mieux les ressources qui sont mises à la disposition de l'application par la machine (temps d'exécution, taille, gestion de la mémoire, ...),

5 Les caractéristiques reprises sont rendues possibles avec un langage de développement tels que C# mais dépendent évidemment du programmeur car, même si les aides au développement deviennent de plus en plus importantes, le rôle central est encore détenu par cet être étrange venu d'ailleurs.

1. La programmation orientée objet (POO)

1.1. Programmation structurée

Avant de parler de programmation orientée objet, il convient de décrire la programmation structurée par ses caractéristiques.

Pour ce faire, il suffit d'examiner sur quoi est basé le travail de l'informaticien. Face à un problème, il examine quelles sont les sorties souhaitées et de quelles entrées il dispose pour les générer. A ce moment, il génère un algorithme destiné à être appliqué à des données. Cet algorithme est destiné à être traduit dans un langage de programmation pour fournir un programme.

Nous avons donc l'équation

Algorithme + Données = Programme.

répondant en fait à la question

Quelles sont les actions que doit effectuer le programme ?

De par sa facilité à être mis en œuvre en ce qui concerne sa logique, proche de la nôtre, cette méthode est habituellement utilisée lors de l'apprentissage d'un langage de programmation.

Ce type de programmation permet une assez grande modularité et, par conséquent, permet l'élaboration de programmes importants sans être obligé de tout redévelopper à chaque fois.

1.2. Améliorations à apporter

Les applications deviennent de plus en plus lourdes à développer, évoluent sans cesse et sont le fruit d'équipes de programmeurs.

La programmation structurée montre alors ses limites.

Malgré des précautions concernant notamment une grande modularité, l'évolution et la maintenance des applications posent quelques problèmes, essentiellement parce que le travail à effectuer concernent de nombreuses procédures.

La réutilisation est aussi une voie vers la création (plus rapide) de meilleurs logiciels. Ainsi, la création de nouvelles applications à partir de composants existants, déjà testés, a toutes chances de produire un code plus fiable et peut se révéler nettement plus rapide et plus économique.

1.3. Pourquoi le software ne suit-il pas le hardware ?

Pour présenter la programmation orientée objet, nous nous reposerons sur l'aspect hardware⁶.

Les concepteurs de matériel informatique sont passés de machines de la taille d'un hangar à des ordinateurs portables légers basés sur de minuscules microprocesseurs. Si nous comparons cette évolution avec celle du monde du logiciel, nous pouvons constater que le hardware repose sur l'assemblage de composants. En effet, les ingénieurs en matériel électronique construisent un système à partir de composants préparés, chacun chargé d'une fonction particulière et fournissant un ensemble de services à travers des interfaces définies. La tâche des concepteurs de matériel est considérablement simplifiée par le travail de leur prédécesseurs.

La programmation orientée objet suit ce mode de fonctionnement et consiste à mettre en place des entités parfaitement définies et utilisables par d'autres applications.

1.4. La programmation orientée objet

Dans programmation orientée objet, il y a *objet* ... Dès lors, il est naturel de décrire ce type de programmation par un de ses fondements : l'objet.

Néanmoins, il convient de présenter d'abord l'abstraction de l'objet, à savoir la classe qui nécessite la modélisation des éléments constituant le domaine sur lequel nous travaillons. Une fois ces éléments modélisés, nous pouvons définir les opérations que nous souhaitons leur appliquer.

Maintenant que nous avons fait connaissance avec les principaux acteurs, nous pouvons fixer le vocabulaire de mise

- une **classe** contient
 - des **attributs** (**données membres**) qui sont en fait les caractéristiques d'état des éléments que la classe modélise,
 - les **méthodes** (**fonctions membres**) qui définissent les opérations qu'il est possible d'appliquer à ces caractéristiques, comme leur attribuer ou prendre connaissance de leur état et modéliser leur comportement (réactions aux sollicitations extérieures, actions sur d'autres objets, ...),
- un **objet** est la création d'un élément appartenant à une classe déterminée et chaque objet créé à partir de la classe possède son identité propre permettant de le distinguer de ses semblables.

Nous pouvons alors réécrire l'équation de développement informatique dans le cadre de la programmation orientée objet :

Méthodes + Données = Objets.

répondant, pour sa part, à la question

Sur quelles données porte ce programme ?

6 Extrait de "Au coeur de ActiveX et OLE", de David Chappel.

1.4.1. Une manière de penser

Comme annoncé ci-dessus, la programmation orientée objet permet de résoudre ou, plutôt, d'améliorer quelques faiblesses de la programmation structurée.

En fait, c'est une adaptation de la logique de programmation à la manière dont nous voyons le monde qui nous entoure. Plutôt que d'adapter le problème soumis au mode de raisonnement informatique, nous pouvons adapter le développement informatique à ce même problème. Cela part du principe que nous sommes entourés d'objets réels (voiture, maison, mobilier, personnes, animaux, ...) et virtuels (salaires, sécurité sociale, ...) que nous allons définir par l'intermédiaire des *classes*.

1.4.1.1. Exemple

Ainsi, nous pouvons associer une voiture à un objet caractérisé par

- la marque et le modèle,
- le type de voiture (standard, break, monovolume, sportive, ...),
- le moteur,
- le nombre de portes,
- les sièges,
- la couleur,
- ...

et certaines de ses caractéristiques peuvent elles-mêmes être associées à des objets comme, par exemple, le moteur pour lequel nous pouvons noter

- le type de carburant,
- la puissance,
- la cylindrée,
- ...

En ce qui concerne les méthodes, nous pourrions définir Démarrer, Rouler, Arrêter, Allumer_Jauge, ...

1.4.1.2. Exemple

Nous pouvons également considérer qu'une maison est un assemblage

- de fondations,
- de murs,
- d'une toiture,
- ...

où nous retrouvons la définition des murs par l'intermédiaire

- de briques et de blocs caractérisés par
 - les dimensions,
 - le matériau,
 - ses caractéristiques hydrofuges et ignifuges,
 - son prix unitaire,
 - la couleur et l'apparence,
 - ...

- de fenêtres caractérisées par
 - les dimensions,
 - le bois utilisé (qui peut être caractérisé par ses propriétés, son prix, le traitement souhaité, ...),
 - la couleur,
 - le carreau associé (qui peut être caractérisé par ses dimensions, son prix, ses caractéristiques d'isolation acoustique et calorifique, ...),
 - ...
 - ...

Signalons aussi que les caractéristiques des portes ne sont guère éloignées de celles des fenêtres, tout comme les portes-fenêtres qui sont un *mélange* des deux.

1.4.1.3. Exemple

L'exemple des animaux correspond bien à cette classification puisque d'éminents scientifiques l'ont effectuée pour nous. Ainsi, partant de la classe de départ, nous pouvons obtenir :

- animaux
 - vertébrés
 - poissons
sélaciens (requin, raie, ...), téléostéens (carpe, anguille, saumon, perche, ...),
dipneustes, ...
 - batraciens
grenouilles, crapauds, tritons et salamandres,
 - reptiles
sauriens (lézards), ophidiens (serpents), chéloniens (tortues) et crocodiliens,
 - oiseaux
palmipèdes, rapaces, gallinacés, colombins, échassiers, grimpeurs, passereaux et coureurs,
 - mammifères
primates, insectivores, chéiroptères, carnassiers, ongulés, cétacés, rongeurs, édentés,
marsupiaux, monotrèmes,
 - invertébrés
 - insectes
 - crustacés
 - mollusques
 - vers
 - oursins
 - ...

Il en est de même pour le règne végétal.

1.4.2. Encourager le recyclage

De par la définition de la programmation orientée objet, il est possible de se consacrer uniquement à un objet et de l'améliorer ou à lui donner de nouvelles fonctionnalités pour, par la suite, les rendre accessibles par les entités qui l'utilisent. Ces mêmes entités ne doivent pas être modifiées car elles voient l'objet amélioré exactement de la même manière et vont seulement constater, espérons-le, que son efficacité est plus grande.

Par exemple, des améliorations concernant le moteur ne modifient pas les caractéristiques de la voiture si ce n'est celles qui sont relatives au ... moteur.

1.5. Les outils disponibles

Nous pouvons maintenant présenter quelles sont les possibilités permettant de gérer ces classes d'objets et, notamment, d'éviter d'inventer la roue lors de chaque développement informatique. En effet, la programmation orientée objet ne serait rien sans les concepts qui suivent qui permettent d'aller au bout de cette conceptualisation particulière.

1.5.1. L'héritage

L'héritage est un concept qui découle des processus cognitifs naturels que sont la classification, la généralisation et la spécialisation.

Précisons quelque peu cette définition. Lorsque nous travaillons en classes, il est tout naturel de disposer des opérations suivantes :

- la classification est le tri hiérarchisé de différentes classes,
- la généralisation permet de lier certaines classes ayant des caractéristiques communes à une classe possédant ces caractéristiques et devenant alors une classe de référence,
- la spécialisation permet de lier une classe avec d'autres classes qui dépendent fonctionnellement de la première mais sont plus spécifiques, plus particulières ou encore plus raffinées.

La nature ou plutôt la classification qu'en a fait l'homme est une parfaite illustration de ces notions. En effet,

- les animaux sont classifiés en différentes espèces,
- les primates, les insectivores, les chéiroptères, les carnassiers, les ongulés, les cétacés, les rongeurs, les édentés, les marsupiaux et les monotrèmes possèdent des caractéristiques communes qui permettent d'être généralisés en mammifères,
- la classe de départ est Animaux qui est spécialisée en vertébrés et invertébrés, elles-mêmes spécialisées en ...

Jusqu'à présent, nous avons développé l'héritage simple : la classe plus spécifique tire son *passé* d'une seule classe.

La programmation orientée objet permet également l'héritage multiple : plusieurs classe permettent de dériver une nouvelle classe plus spécifique.

Pour illustrer nos propos, passons à un autre exemple qui est celui de l'aviation. Partant d'une classe générale qui est celle des avions, nous pouvons affiner notre modélisation comme suit :

- avion
 - avion commercial
 - transport de passagers
 - transport de marchandises
 - avion militaire
 - avion de combat
 - avion de transport
 - avion de reconnaissance
 - avion civil

Cette manière de voir les choses ne concerne que la partie vol des objets définis. En effet, en ce qui concerne l'avion militaire, par exemple, il est possible de lui adjoindre une autre caractéristique relative à son aspect *guerrier* (pouvant d'ailleurs être utilisé par tout autre matériel à vocation militaire) : l'armement. Cette spécificité peut être modélisé indépendamment et être ajoutée, en phase finale, pour caractériser les avions militaires de combat.

Sans entrer dans les détails, nous pouvons également reprendre l'exemple du règne animal où la classe des omnivores peut être considérée comme héritière des herbivores et des carnivores.

1.5.2. L'encapsulation

Ce processus permet de protéger la structure et, en particulier, les données de la classe. Ce qui ne doit pas nécessairement être accessible est alors caché et, si nécessaire, l'accès est rendu possible par l'intermédiaire des méthodes associées à la classe.

La vie de tous les jours nous permet à nouveau d'illustrer cette fonctionnalité. Lorsque nous augmentons le son de notre télévision, il ne nous est pas nécessaire de d'accéder à l'intérieur de notre poste pour modifier l'état des composants qui gèrent le son mais, simplement, d'actionner le bouton adéquat. Il est même très sécurisant, pour la bonne santé de notre télévision, que nous ne puissions pas accéder à son *intérieur*.

La programmation orientée objet met en fait à notre disposition trois niveaux de sécurité :

- **publique** : les données et les fonctions ainsi caractérisées sont accessibles par toute autre classe,
- **protégé** : les données et les fonctions qualifiées de la sorte sont rendues accessibles à toute classe héritière,
- **privé** : les données et les fonctions ainsi caractérisées ne peuvent être accessibles que par la classe elle-même.

1.5.3. Le polymorphisme

Le nom de *polymorphisme* vient du grec et signifie *qui peut prendre plusieurs formes*. La programmation orientée objet adapte ce comportement en donnant aux développeurs la possibilité d'utiliser, sous le même nom, plusieurs fonctions différentes par les paramètres qui leur sont transmis (en nombre et/ou en type) aussi bien que par la réponse qu'elle peuvent fournir. La bonne fonction est choisie au moment de l'appel en fonction de ce qui est transmis comme paramètres.

On parle également de surcharge de fonctions.

Toujours en prenant l'exemple des animaux, nous pouvons définir une méthode associée au déplacement de ce dernier qui prend la même dénomination mais qui varie en fonction du type d'animal concerné (voler, nager, marcher, sauter, ramper, ...).

Dans le cadre du mouvement, nous pouvons également considérer une méthode mouvement dans un jeu d'échec qui pourra, grâce au polymorphisme, effectuer le mouvement approprié d'une pièce grâce au type de pièce qui lui sera fourni en paramètre.

1.6. Conséquences sur la modélisation

Avec les outils présentés ci-dessus, nous pouvons définir des entités de manière hiérarchique en partant d'une définition de base et en la précisant de plus en plus pour en arriver à l'objet *final*.

A partir de ce moment, le point de départ de toute modélisation est la description des données sur lesquelles on travaille ainsi que des traitements qui leur sont spécifiques.

Cela permet de réaliser des modules complets au niveau définition et traitements associés qui peuvent être testés aisément et indépendamment puis mis à la disposition de toute autre application. Dans le même ordre d'idée, une modification de la programmation ne modifie pas le comportement des applications qui en dépendent à condition que l'on respecte la nomenclature mise en place.

Notons enfin qu'il existe des méthodes qui permettent de systématiser l'analyse d'un projet informatique selon une approche orientée objet. Comme toute méthode d'analyse fonctionnelle qui se respecte, la modélisation objet consiste à créer une représentation informatique des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de l'implémentation future, c'est-à-dire *indépendamment d'un langage de programmation*.

Pour ce faire, des méthodes ont été mises au point. Entre 1970 et 1994, de nombreux analystes ont mis au point des approches orientées objets, si bien qu'il en existait plus de 50 méthodes objet parmi lesquelles

- La méthode OMT de Rumbaugh
- La méthode BOOCH'93 de Booch
- La méthode OOSE de Jacobson

A partir de 1994, Rumbaugh et Booch (rejoints en 1995 par Jacobson) ont unis leurs efforts pour mettre au point la méthode UML (Unified Modeling Language), qui permet de définir une notation standard en incorporant les avantages de chacune des méthodes précédentes (ainsi que celles d'autres analystes). Cette méthode est devenu désormais la référence en terme de modélisation objet, à un tel point que sa connaissance est souvent nécessaire pour obtenir un poste de développeur objet.

1.7. Et au niveau informatique

Dans la programmation actuelle, il devient obligatoire de développer dans un environnement de type Windows avec fenêtre et événements car cela permet de donner un look professionnel à vos applications, conformément aux développements effectués par la concurrence.

Comme annoncé, nous sommes alors en plein dans la programmation orientée objet. Evidemment, il est possible d'étendre cette manière de programmer à l'ensemble de son travail même si la programmation structurée coexiste avec les classes, méthodes et autres objets.

Examinons maintenant quelles sont les avantages de ce type de programmation, dans un environnement comme celui de Windows.

1.7.1. Les objets : identification

Il s'agit d'entités disponibles directement dans l'environnement choisi comme les fenêtres, les contrôles (zones d'encodage, listes déroulantes, boutons à cocher, ...), les menus, les barres d'outils, les barres de défilement, ...

Ces objets peuvent effectuer des opérations spécifiques comme s'afficher, analyser l'information reçue, réagir aux manipulations (en vérifiant et corrigeant), envoyer de l'information à d'autres objets (la barre de défilement demande à la fenêtre de modifier la vue affichée, ...).

1.7.2. Les objets : rôle

Les objets peuvent se promener, parler et écouter, et encore bien d'autres choses !

Ces actions sont effectuées par l'intermédiaire de méthodes. Par celles-ci, il est possible de déplacer des objets, de les faire parler ou transmettre de l'information par une méthode d'accès à certains de ses composants (publics), de les faire écouter par l'intermédiaire d'une méthode d'assignation de certains de ses composants.

Et ils peuvent faire bien d'autres choses ...

1.7.3. Les objets : définition

La définition des objets se retrouve dans les classes. Il s'agit de textes de programmes reprenant les attributs et leur description ainsi que les méthodes et la programmation (orientée objet et ... structurée) de leurs actions.

Certaines classes sont livrées en standard par le support de développement utilisé. Elles sont utilement complétées par Windows et ses célèbres (mais difficiles d'accès) API (Application Programming Interface).

Ces classes dérivent, pour la plupart, d'autres classes dont elles héritent d'un certain nombre de caractéristiques.

Ainsi, la gestion des fenêtres de dialogue peut se caractériser ainsi :

- une classe *événement* permettant de gérer les événements pouvant survenir (clics de souris, horloge interne, arrivée de courrier, dialogue avec l'imprimante, ...),
- une classe *fenêtre* basique héritant de la gestion des événements et ajoutant des attributs (fond, couleur, titre, menu, ...) et des méthodes appropriées,
- une classe *fenêtre de dialogue* reprenant les caractéristiques d'une fenêtre standard et y adjoignant des possibilités spécifiques à ce genre de fenêtre (notamment en ce qui concerne le dialogue avec l'application : paramétrage à la création et renvoi d'une valeur de réponse).

1.7.4. Les objets : gestion

Les classes sont déclarées dans le programme en tant que tel. Par contre, les objets sont créés lors de l'exécution de ce même programme, à la demande de ce dernier (que cela provienne de l'utilisateur ou du dialogue entre le programme et le système d'exploitation). Il en est de même pour la destruction.

Nous comprenons dès lors que certaines méthodes jouent un rôle particulier. Il s'agit des constructeurs destinées à mettre en place l'objet, à partir de la définition trouvée dans la classe et des destructeurs qui sont destinés à faire place nette une fois l'objet devenu inutile (par l'utilisateur qui ferme la fenêtre ou par le système qui, voyant la fenêtre détruite, ferme également le menu, les barres de défilement, les contrôles, ... associés à la fenêtre).

La création d'un objet est baptisée *instanciation*.

Notons également que, tout comme des variables de type classique, les objets en provenance de la même classe sont identiques de par leur structure mais possèdent une identité propre permettant d'accéder l'un ou l'autre.

1.7.5. Les objets : conséquences événementielles

Par l'intermédiaire de la programmation orientée objet, il est possible d'accéder à la programmation événementielle et de modifier l'interaction entre l'utilisateur et le logiciel.

Avant, l'utilisateur interagissait avec le logiciel lorsque ce dernier le lui demandait et la liberté n'était pas de mise. Le programme ouvrait des fenêtres d'encodage ou de *dialogue* et l'utilisateur pouvait indiquer ses choix. Maintenant, avec la programmation événementielle, les choses sont différentes. L'exemple le plus simple consiste à prendre une fenêtre d'encodage des données. Avant, l'ordre d'encodage était dicté par l'ordinateur. Maintenant, il est possible de modifier, par l'intermédiaire de la souris, l'ordre d'encodage. Il est même possible de passer à une autre application, ouvrir et consulter une autre fenêtre, ...

2. L'environnement .NET

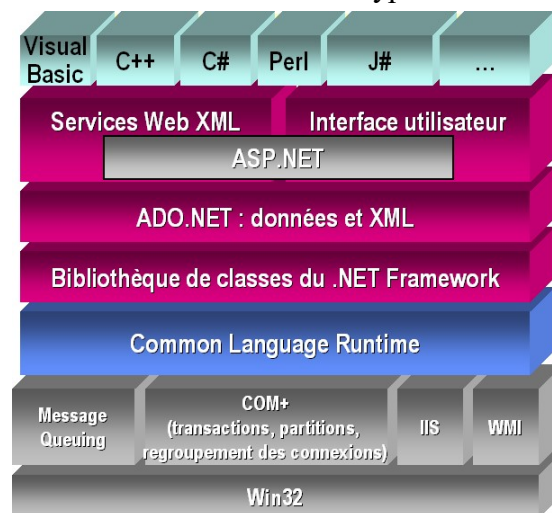
Avant de nous plonger dans l'étude du langage de programmation C#, nous allons commencer par présenter la plate-forme de développement .NET qui fournit le canevas de compilation et d'exécution nécessaire à la création d'applications.

Cette dernière se base sur une nouvelle stratégie de répartition de l'information et de son traitement en s'appuyant principalement sur

- une disparition progressive des différences entre les applications et Internet, les serveurs fournissant en plus des pages HTML des services à des applications distantes,
- les informations peuvent être réparties sur plusieurs machines proposant chacune un service adapté aux informations qu'elles détiennent,
- à la place d'une seule application, l'utilisateur a accès à un ensemble d'applications pouvant coopérer entre elles,
- l'utilisateur loue des services à la place d'acheter des logiciels.

Les fondations de cette architecture reposent sur un environnement d'exécution des applications .NET : .NET Framework. A cet outil s'ajoute Visual Studio .NET qui contient l'environnement RAD de développement pour l'architecture .NET.

Microsoft se base habituellement sur un schéma du type



Reprenons les différentes couches :

- les langages conformes à la norme CLS⁷ (Visual Basic, Visual C++, Visual C#, Visual J#, JScript .NET) dont certains sont extérieurs à Visual Studio .NET (Cobol, Delphi, ...)
- l'interface (graphique) utilisateur (fenêtres windows, formulaires Web, services Web et applications console) avec une place particulière pour ASP.NET,
- la couche suivante prend en charge XML et une nouvelle génération de la technologie ADO (Activex Data Object),

⁷ CLS signifie Common Language Signification et reprend les caractéristiques devant être respectées par les langages pris en charge par l'architecture .NET. Nous pourrions ajouter cette couche juste en dessous des langages supportés par .Net.

- une bibliothèque de plusieurs centaines de classes facilitant le développement,
- le Common Language Runtime (CLR) fournit un environnement managé (certains services fournis automatiquement) d'exécution robuste et sécurisé, prend en charge plusieurs langages de programmation tout en simplifiant le déploiement et la gestion des applications,
- l'environnement .NET est conçu pour fonctionner sur les systèmes d'exploitation Win32 (et Windows CE).

2.1. Manipulations en dehors de Visual Studio .NET

Pour illustrer ce qui suit, nous allons considérer, sans entrer dans les détails, le classique source suivant, déjà rencontré en C, que nous supposons avoir enregistré dans le fichier `monprog.cs`. Nous reprendrons ce fichier C# pour découvrir l'environnement .NET.

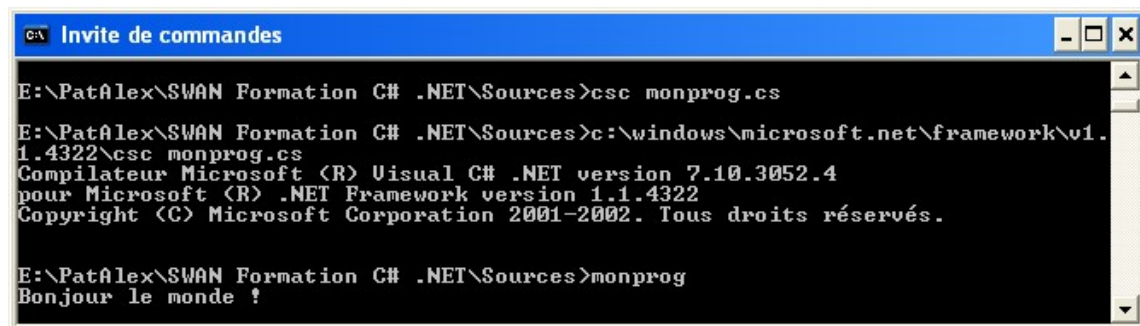
```
class prog
{
    static void Main()
    {
        System.Console.WriteLine("Bonjour le monde !");
    }
}
```

En mode console (fenêtre DOS), nous pouvons, pour compiler et créer un fichier *exécutable*, utiliser la commande⁸ sous la forme

```
csc monprog.cs
```

Le fichier portant le même nom mais d'extension `exe` est alors généré. Il convient de noter que ce fichier est un faux exécutable car il nécessite la présence d'une machine virtuelle pour être effectivement exécuté par la machine. Le fichier est composé d'une première traduction des instructions écrites en C# dans le Microsoft Intermediate Language (MSIL).

Histoire de s'extasier devant le résultat de ce premier chef d'œuvre, nous pouvons, toujours en ligne de commande, exécuter le programme en tapant `monprog`. La fenêtre suivante reprend les diverses commandes effectuées :



```

C:\> Invite de commandes

E:\PatAlex\SWAN Formation C# .NET\Sources>csc monprog.cs

E:\PatAlex\SWAN Formation C# .NET\Sources>c:\windows\microsoft.net\framework\v1.1.4322\csc monprog.cs
Compilateur Microsoft (R) Visual C# .NET version 7.10.3052.4
pour Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.

E:\PatAlex\SWAN Formation C# .NET\Sources>monprog
Bonjour le monde !
```

⁸ Le programme `csc.exe` ne se trouve pas dans le répertoire de travail. Par conséquent, une fois identifié le répertoire où cet exécutable se trouve, trois possibilités s'offrent à vous :

- ajouter ce répertoire dans le path déjà défini,
- appeler la commande `csc` en spécifiant le chemin d'accès,
- créer un fichier batch de commande DOS où se trouve l'instruction

```
csc %1
```

la commande `csc` étant précédée du chemin d'accès.

2.2. Manipulation à l'aide de Visual Studio .NET

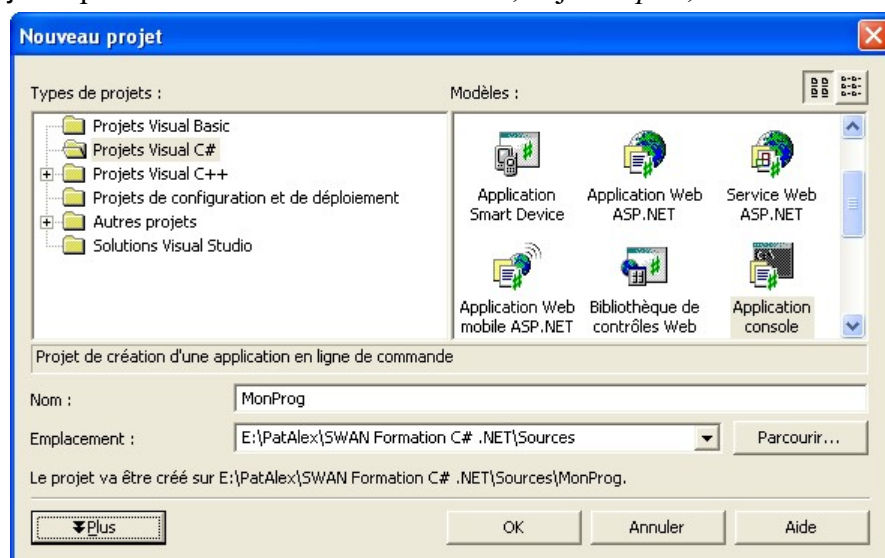
Même si ces procédures *archaïques* sont parfois incontournables, nous n'allons pas refuser le *progrès* et nous pencher sur la découverte de l'environnement Visual Studio .NET qui est un environnement interactif de développement (IDE). Notre tâche va être facilitée grâce à

- un éditeur de code interactif,
- une aide disponible,
- un outil graphique de création de fenêtre,
- un compilateur et un débogueur intégré.

2.2.1. Créer une application

Nous allons commencer en douceur par la création d'un *projet*⁹ d'application console et essayer d'y insérer le code présenté auparavant.

Une fois Visual Studio .NET chargé, nous nous rendons dans le menu Fichier et y pointons Nouveau, Projet ... pour découvrir la fenêtre suivante, déjà *remplie*,

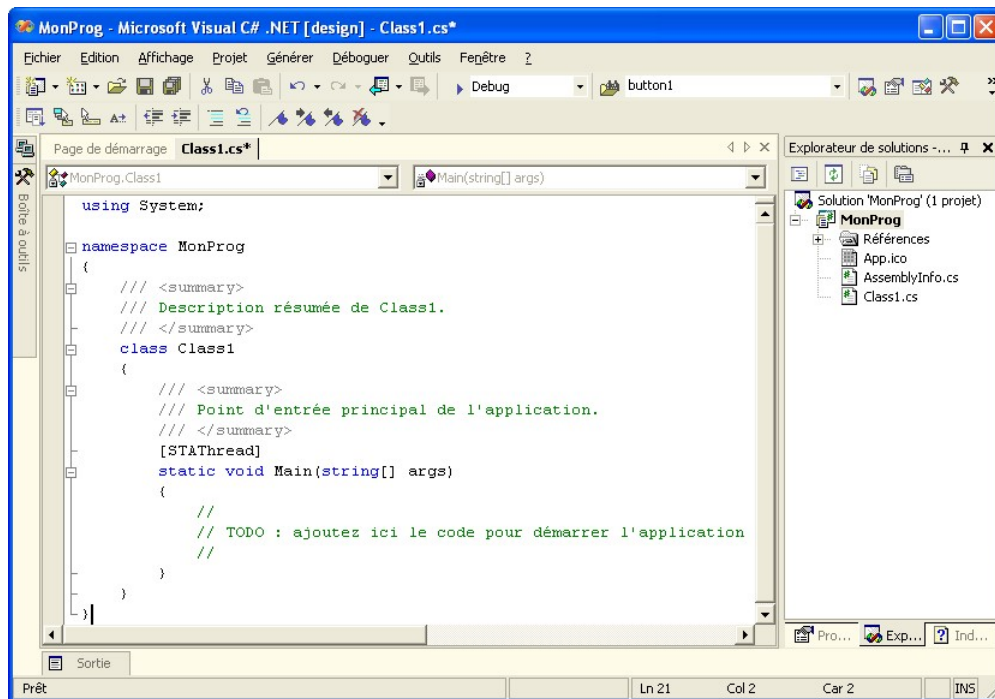


Les renseignements à donner sont les suivants :

- dans Types de projets :, sélectionner Projets Visual C#,
- dans Modèles :, sélectionner Application console,
- dans Nom :, renseigner le nom attribué au projet, MonProg dans ce cas,
- dans Emplacement :, déterminer le répertoire où stocker le projet et ses différentes ressources.

⁹ Le fait de travailler dans un environnement de développement entraîne l'adoption du vocabulaire utilisé. Ainsi, on ne parlera plus de programme ou d'application mais plutôt de projet, terme qui englobe plus qu'un logiciel, comme nous le découvrirons au fur et à mesure.

Une fois ces choix validés, la fenêtre suivante (ou à peu près) nous apparaît :



Profitions de cette fenêtre pour signaler

- la présence d'un menu et de barres d'outils,
- le programme est enregistré dans le fichier Class1.cs (* signifiant que les modifications n'ont pas encore été sauvegardées),
- nous modifions ce nom de fichier en activant Enregistrer Class1.cs sous ... du menu Fichier et en renseignant Bienvenu.cs,
- le remplacement de la directive #include chère à C semble remplacée par l'instruction `using`,
- le nom MonProg choisi auparavant apparaît sous la clause `namespace`,
- la classe sur laquelle nous nous sommes appuyés jusqu'à présent s'appelle Class1 (à modifier en dactylographiant Bienvenue) et est incluse dans l'espace de noms,
- des commentaires *particuliers* agrémentent le code source et permettent une documentation *automatique*,
- le mot clé STAThread qui indique que le modèle de thread (processus, application en cours d'exécution) est un modèle STA (single-threaded apartment),
- la méthode Main() est prête à recevoir des arguments en ligne de commande par l'intermédiaire de args,
- l'indication bien claire de l'endroit où nous pouvons taper notre code, à savoir `Console.WriteLine("Bonjour le monde");`
- des fenêtres déroulantes (ou non) disponibles (ou pouvant le devenir via le menu Affichage) de part et d'autre du code présenté en partie centrale :
- un Explorateur de serveurs et une Boîte à outils à gauche,
- les Propriétés, l'Explorateur de solutions et l'Index de l'aide à droite.

Nous préférons, dans ces notes, nous pencher sur le langage C# en tant que tel en utilisant les ressources directement utiles. Nous allons donc, dans ce qui suit, effleurer différents outils et laissons le soin à l'étudiant d'approfondir cette étude au fur et à mesure de son travail.

2.2.2. Génération et affichage de la documentation XML du code source

Notre premier contact avec les outils de Visual Studio .NET peut se faire par l'intermédiaire de la documentation du code source.

Cet utilitaire est fourni avec l'IDE et se base sur les commentaires de type XML que le développeur place au sein du source de son application.

Pour accéder à cet outil, il suffit d'activer le point Générer les pages Web de commentaires ... du menu Outils. Le choix est alors laissé au développeur de savoir ce qu'il souhaite *éditer* et où le résultat doit être enregistré.

Dans le cas de notre manipulation, le résultat est le suivant



2.2.3. Compilation et exécution d'un programme C#

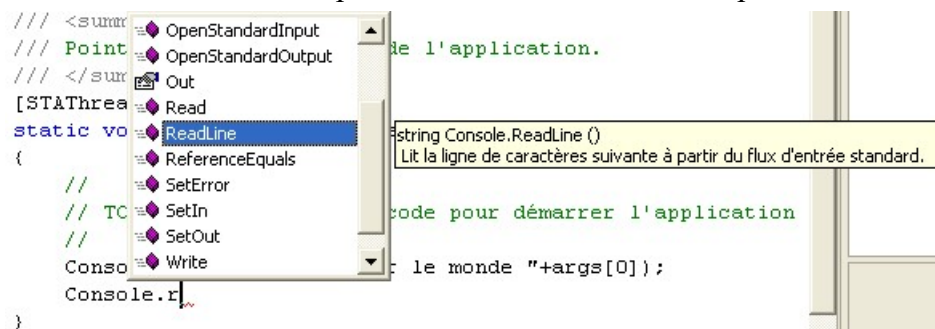
Avant de compiler le programme, nous allons commencer par ajouter une commande de lecture au clavier permettant d'effectuer, lors de l'exécution, un arrêt sur écran. La fonction `Main()` devrait alors se mettre sous la forme

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

La première chose à constater est que nous sommes espionnés lors de la dactylographie de nos instructions (technologie IntelliSense de Visual Studio .NET). En effet, le fait de taper

`Console.r`

fait apparaître tout une série de choix possibles avec une brève description de chacun d'eux :

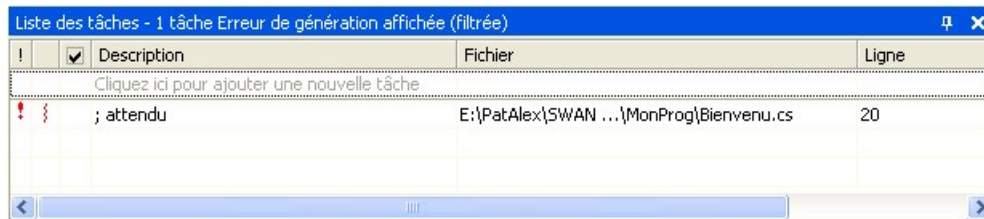


Cela ne se termine pas là puisque, une fois notre commande tapée, un petit signe fait son apparition en fin de ligne :


```
Console.ReadLine()~
```

La compilation de ce petit programme nous éclaire directement sur la signification de ce petit symbole.

Le fait d'activer, dans le menu Générer, le choix Générer MonProg, nous affiche le morceau de fenêtre



Evidemment, sommes-nous stupides ! Nous avons oublié de renseigner la fin de l'instruction par le ;, comme en C. Une fois cette erreur impardonnable corrigée, nous recevons le message de réussite de la génération.

Il nous suffit alors d'exécuter le programme par l'intermédiaire de Démarrer dans Déboguer (choix qui peut être remplacé par le bouton  de la barre d'outils ou par la touche **F5**). Nous obtenons alors



2.2.4. Utilisation du débogueur

La compilation ne permet pas d'assurer la validité d'un programme. Parfois, les erreurs se cachent jusqu'à l'exécution du programme.

Pour illustrer ce problème, nous pouvons travailler sur la fonction `Main()` suivante

```
static void Main()
{
    int taille;
    float pideal;
    Console.WriteLine("Encoder votre taille : ");
    taille = Int32.Parse(Console.ReadLine());
    pideal = 50+3*(taille-150)/4;
    Console.WriteLine("Poids ideal : ");
    Console.WriteLine("  Homme : "+pideal);
    Console.WriteLine("  Femme : "+0.9*pideal);
    Console.ReadLine();
}
```

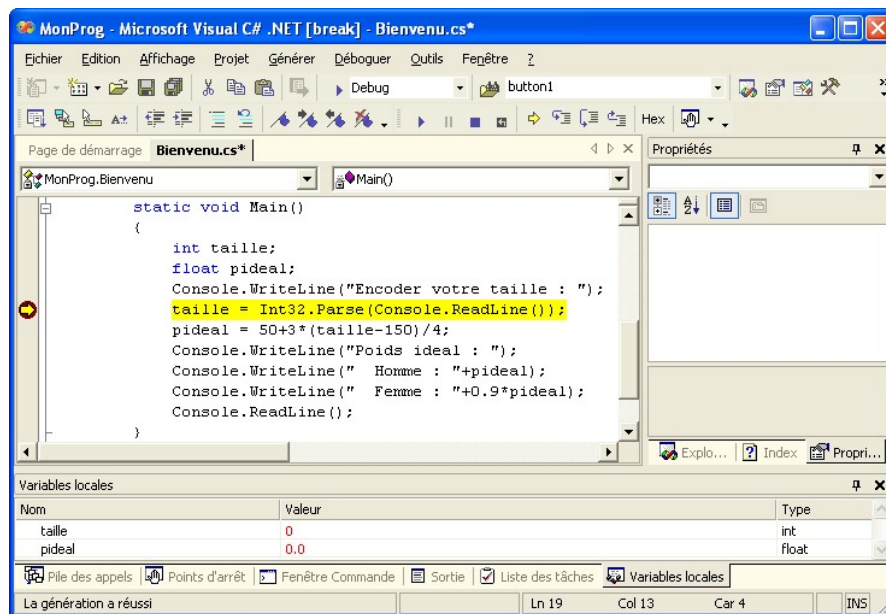
Si nous encodons la valeur 180 en tant que taille de l'individu, nous sommes confrontés aux réponses 72 et 64,8. Cependant, nous aurions dû obtenir 72,50 comme première valeur, la deuxième s'adaptant à cette valeur et devant être 65,25. Pour pister les calculs, nous pouvons nous appuyer sur le débogueur.

Nous pouvons cliquer avec le bouton droit de la souris sur la ligne

```
taille = Int32.Parse(Console.ReadLine());
```

et choisir Insérer un point d'arrêt. Nous lançons alors l'exécution du programme comme précédemment.

Visual Studio .NET. stoppe cette exécution sur la ligne renseignée, les variables propres à la fonction étant affichée dans une fenêtre auxiliaire. La situation peut être la suivante :



Nous constatons que

```
taille 0
pideal 0.0
```

Ensuite, nous demandons à passer à l'instruction suivante par le choix Pas à pas principal du menu Déboguer. Il s'agit alors d'encoder la valeur 180 au clavier. L'espionnage en cours nous renseigne sur

```
taille 180
pideal 0.0
```

et l'instruction suivante nous fournit

```
taille 180
pideal 72.0
```

Nous savons maintenant que l'erreur se situe au niveau du calcul de la variable `pideal`. La faute est classique : nous stockons dans une variable réelle un calcul effectué sur des entiers. Malgré le type de la variable, le résultat du calcul est entier ...

La correction peut être

```
pideal = 50+(float)3*(taille-150)/4;
```

et ... ça fonctionne (évidemment).

Pour traquer les erreurs de fonctionnement d'un programme, C# sous .Net nous donne également des outils pour effectuer des vérifications plus fines.

Nous allons commencer par la méthode `Assert()` de la classe `Debug` définie dans l'espace de noms `System.Diagnostics`. Elle nous permet de fournir des renseignements sur les problèmes rencontrés à l'exécution.

Supposons que nous souhaitions développer un petit programme qui calcule la somme des `n` premiers nombres entiers, `n` étant un entier positif encodé par l'utilisateur. Nous pouvons écrire

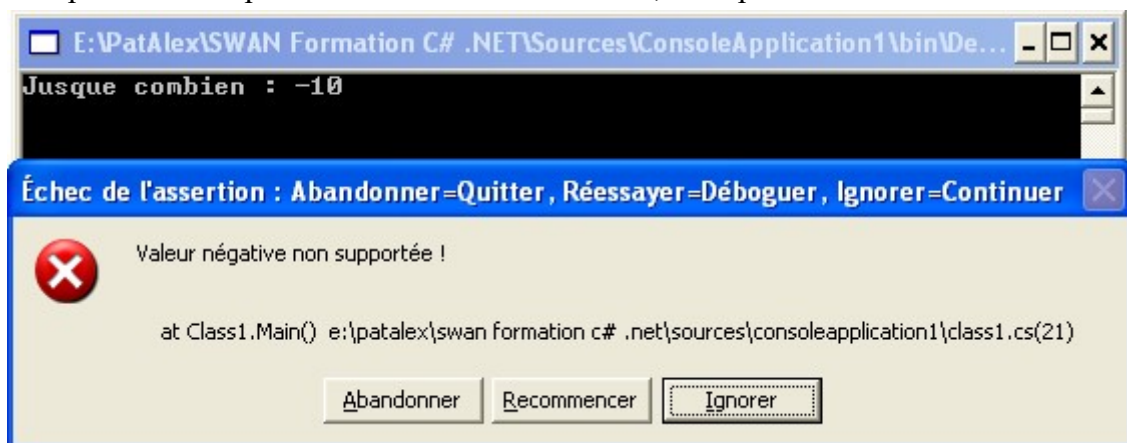
```
static void Main()
{
    int i, som=0, n;
    Console.WriteLine("Somme des ... premiers nombres : ");
    n = Int32.Parse(Console.ReadLine());
    for(i=1; i<=n; i++) som+=i;
    pideal = 50+3*(taille-150)/4;
    Console.WriteLine("Valeur obtenue : "+som);
    Console.ReadLine();
}
```

Le problème est que, pour une valeur négative, nous obtenons une valeur nulle sans aucune explication tandis que, une fois la valeur 65535 dépassée, la valeur obtenue est pour le moins fantaisiste, pouvant même devenir négative¹⁰.

Pour rendre notre programme plus robuste, nous pouvons insérer, en renseignant préalablement la référence à l'espace de noms souhaité, deux instructions permettant de renseigner l'utilisateur sur le comportement réel de l'application. Ainsi, avant de lancer le calcul par l'intermédiaire de la boucle `for()`, nous pouvons insérer les instructions

```
Debug.Assert(n>=0, "Valeur négative non supportée !");
Debug.Assert(n<=65535, "Valeur supérieure à 65535 non supportée !");
```

qui affichent le message renseigné en second argument lorsque la condition rencontrée en première position n'est pas vérifiée. Lors de l'exécution, nous pouvons avoir



¹⁰ Ce type de comportement provient de dépassement de capacité ou plus exactement, dans ce cas-ci, de valeurs limites pour les entiers. Nous verrons ultérieurement que de tels dépassements peuvent donner lieu à un message d'erreur plutôt qu'à un comportement d'apparence normale.

La gestion des erreurs peut également se faire de manière personnalisée et être intégrée au sein même du programme. En effet, il n'y a rien de plus désagréable qu'un message d'erreur venant du système et complètement incompréhensible. Par conséquent, les exceptions rencontrées peuvent être prises en charge par le développeur. Nous y revenons en 0.

2.2.5. Utilisation du désassembleur du langage intermédiaire

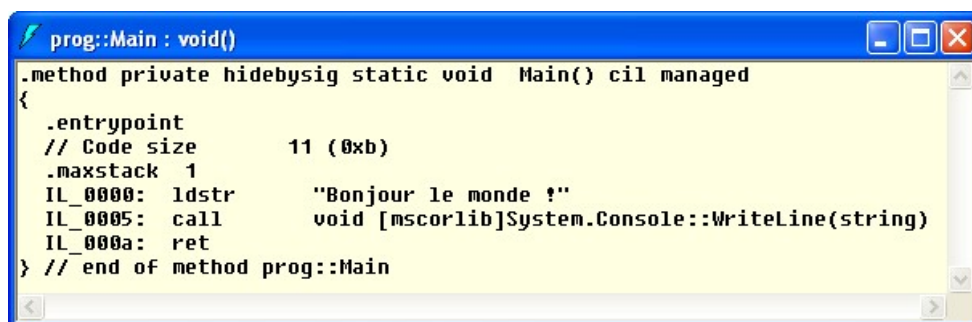
Le lecteur curieux pourra examiner le contenu d'un fichier d'extension `exe` généré par Visual Studio .NET. Pour rappel, il s'agit de la traduction de notre source en en MSIL (proche de l'assembleur).

Il est possible de retourner en ligne de commande par l'intermédiaire d'une fenêtre DOS. Dans ce cas, une fois identifié l'endroit de stockage du programme `ildasm.exe`, nous pouvons lancer la commande

```
ildasm monprog.exe
```

moyennant, par exemple, la copie des fichiers `ildasm.exe` et `ildasm.config.exe` dans le répertoire où se trouve `monprog.exe`.

Les deux vues suivantes se focalisent sur l'utilisation de l'utilitaire `ildasm` :



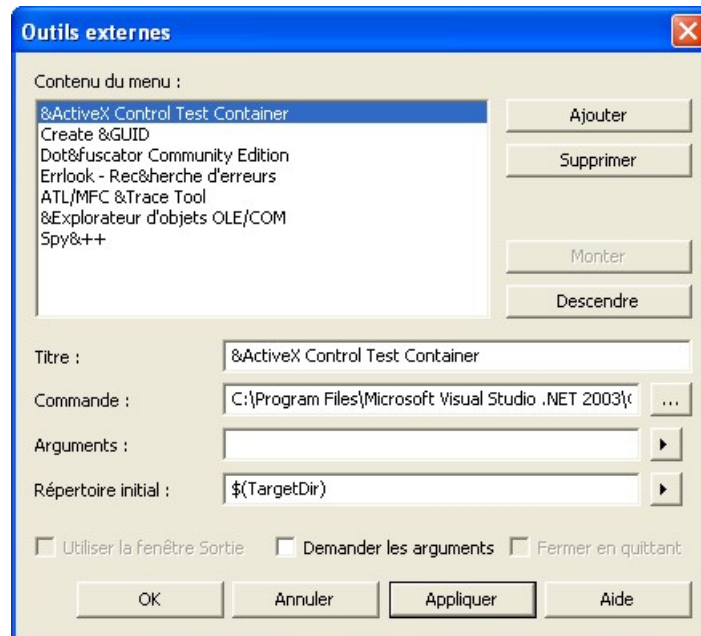
Nous pouvons ainsi constater que la machine exécute les instructions

- chargement d'une chaîne de caractères `Bonjour le monde !`,
- appel de la méthode `WriteLine()` de la classe `Console` enregistrée dans `System` avec, en paramètre, un `string` (chaîne de caractères),
- fin du parcours avec l'instruction de retour `ret`.

Signalons également qu'un double-clic sur le point `MANIFEST` permet de découvrir divers renseignements sur l'application que nous venons de créer dont le numéro de version.

Pour terminer ce paragraphe, nous pouvons effectuer les manipulations précédentes directement depuis l'environnement Visual Studio .NET, il nous *suffit* de l'intégrer dans les outils.

Dans le menu Outils, il existe une option Outils externes. Une fois ce choix activé, nous sommes confrontés à la fenêtre suivante



Une fois enfoncé le bouton Ajouter, il nous suffit de remplir les divers renseignements souhaités, à savoir

- **Titre** : Désassembleur, par exemple,
- **Commande** : renseigner `ildasm.exe`, en parcourant le disque dur pour intégrer le chemin d'accès,
- **Arguments** : `$(TargetDir)\$(TargetName).exe` ce qui peut s'obtenir en ajoutant les caractères manquants aux choix **Répertoire cible** et **Nom de la cible**.

Les boutons **Monter** et **Descendre** permettent de placer le nouveau point de menu dans la *hiérarchie*.

Une fois validées ces informations, un nouveau choix s'est ajouté dans les options possibles du menu Outils, à savoir Désassembleur.

Pour information, nous pouvons même associer un raccourci clavier lors du déroulement du menu en renseignant `D&ésassembleur` comme **Titre**. Nous verrons alors apparaître **Désassembleur** dans le menu Outils.

L'environnement de développement .Net permet une analyse du code directement depuis celui-ci. Nous disposons de l'espace de noms `System.Reflection` qui nous permet de recueillir des informations quant à l'application en cours. Bien que cela sorte quelque peu du cadre de ce cours, dans l'état actuel des choses, nous présentons ci-après un programme qui effectue ce type d'analyse.

Nous pouvons écrire

```

using System;
using System.Reflection;

class PresentationString
{
    static void Main(string[] args)
    {
        Type t = Type.GetType("System.String");
        Console.WriteLine("Nom : "+t.FullName);
        Console.WriteLine("Assemblage : "+t.AssemblyQualifiedName);
        Console.WriteLine("GUID : "+t.GUID);
        Console.WriteLine("Abstrait ? : "+t.IsAbstract);
        Console.WriteLine("Classe ? : "+t.IsClass);
        Console.WriteLine("Interface ? : "+t.IsInterface);
        Console.WriteLine("Non base ? : "+t.IsSealed);
        ConstructorInfo[] ci = t.GetConstructors();
        Console.WriteLine("Constructeurs : "+ci.Length);
        foreach(ConstructorInfo c in ci)
        {
            Console.WriteLine(" - "+c);
            ParameterInfo[] pi = c.GetParameters();
            Console.WriteLine(" ");
            foreach(ParameterInfo p in pi)
            {
                Console.WriteLine(p.Name+" ");
            }
            Console.WriteLine();
        }
        Console.ReadLine();
    }
}

```

Le résultat à l'écran de cette *introspection* est

```

E:\PatAlex\SWAN Formation C# .NET\Sources\ConsoleApplication2\bin\Debug\ConsoleAppli...
Nom : System.String
Assemblage : System.String, mscorlib, Version=1.0.5000.0, Culture=neutral, Pu
blicKeyToken=b77a5c561934e089
GUID : 296afbff-1b0b-3ff5-9d6c-4e7e599f8b57
Abstrait ? : False
Classe ? : True
Interface ? : False
Non base ? : True
Constructeurs : 8
- Void .ctor(Char*)
    value,
- Void .ctor(Char*, Int32, Int32)
    value, startIndex, length,
- Void .ctor(SByte*)
    value,
- Void .ctor(SByte*, Int32, Int32)
    value, startIndex, length,
- Void .ctor(SByte*, Int32, Int32, System.Text.Encoding)
    value, startIndex, length, enc,
- Void .ctor(Char[], Int32, Int32)
    value, startIndex, length,
- Void .ctor(Char[])
    value,
- Void .ctor(Char, Int32)
    c, count,

```

3. Généralités

Histoire de ne pas encombrer les codes sources avec les ressources nécessaires à une programmation Windows, nous allons présenter les notions de base associées à C# dans un environnement de développement de type *console*.

3.1. Le premier programme C#

3.1.1. Programme minimum

Le programme minimum que nous puissions écrire en C# est le suivant :

```
class MonProg
{
    static void Main()
    {
    }
}
```

Le point d'entrée habituel `main()` de C est remplacé par `Main()` et est maintenant encapsulé dans une classe de notre cru, `MonProg` en l'occurrence.

3.1.2. Programme fournissant une réponse

La communication entre programmes est toujours possible puisque notre fonction, ou plutôt méthode, peut renvoyer une réponse à son *père*. Ainsi, nous pouvons écrire le programme suivant qui, d'ailleurs, ne fait pas plus que le précédent,

```
class MonProg
{
    static int Main()
    {
        return 0;
    }
}
```

3.1.3. Programme recevant des renseignements

Il est également possible pour un programme de recevoir des données de son *père* comme dans le programme suivant

```
class MonProg
{
    static void Main(string[] args)
    {
    }
}
```

La gestion de données transmises est similaire à ce qui se faisait en C mais est facilité par la transmission via un tableau de chaînes de caractères et non plus un tableau de pointeurs vers des chaînes de caractères.

3.2. Premier contact avec C#

3.2.1. Les espaces de noms

Nous allons commencer par utiliser la procédure `WriteLine()` qui permet d'afficher à l'écran le message passé en argument. Cette procédure est définie dans l'environnement .NET dans l'espace de noms `System` que nous devons spécifier. Ces ressources sont partitionnées en classes et la méthode `WriteLine()` est, naturellement, reliée à la classe `Console`. Ce système est comparable aux fichiers `#include` propres à C.

Pour nous conformer au langage .NET, nous parlerons, pour `System`, d'un espace de noms. L'appel à ces ressources se fait par l'intermédiaire de la commande

```
■ using System;
```

placée en début de code.

Nous pouvons écrire la version C# du programme affichant *Hello World !* :

```
■ using System;
■
■ class MonProg
■ {static void Main()
■     {Console.WriteLine("Hello World !");
■     }
■ }
```

3.2.2. Manipulations sur les espaces de noms

Nous pouvons éviter l'appel de la ressource `System` en la spécifiant lors de l'appel de la méthode. Ainsi, le source suivant est équivalent à ce qui précède sauf que nous avons ajouté la commande `ReadLine()` afin de pouvoir *admirer* le résultat de notre travail.

```
■ class MonProg
■ {static void Main()
■     {System.Console.WriteLine("Hello World !");
■     System.Console.ReadLine();
■     }
■ }
```

Notons également qu'il est possible de renommer les ressources par l'instruction

```
■ using MonEcran = System.Console;
■
■ class MonProg
■ {static void Main()
■     {MonEcran.WriteLine("Hello World !");
■     MonEcran.ReadLine();
■     }
■ }
```

mais nous ne pouvons que déconseiller cette pratique qui peut nuire à la lisibilité du code, essentiellement en cas de partage.

3.2.3. Les entrées-sorties en mode console

Pour terminer avec ce paragraphe, signalons que C# met également les méthodes `Write()` et `Read()` à notre disposition, similaires aux méthodes `WriteLine()` et `ReadLine()` que nous venons de voir.

En consultant l'aide intégrée, nous avons

- `WriteLine()` : écrit dans le flux de sortie standard les données spécifiées, suivies du terminateur de ligne active (méthode surchargée),
- `Write()` : écrit les informations spécifiées dans le flux de sortie standard (méthode surchargée),
- `ReadLine()` : lit la ligne de caractères suivante à partir du flux d'entrée,
- `Read()` : lit le caractère suivant à partir du flux d'entrée standard.

Les flux peuvent être redéfinis.

En ce qui concerne les méthodes d'affichage, il est possible de formater les sorties, comme c'est le cas pour la fonction `printf()` de C, selon le même principe :

```
■ System.Console.WriteLine(string chaîne,object[] arguments); ■
```

où `chaîne` est le texte à afficher, contenant d'éventuels formatage de données à écrire à l'écran et `arguments` est la liste des données intervenant dans l'affichage de `chaîne`.

En se basant sur ce qui précède, nous pouvons placer dans `chaîne` un formatage du type `{n,m:s}` où

- `n` est la position de la donnée à afficher dans la liste `argument`,
- `m` est le nombre d'espaces réservés pour afficher la donnée, une valeur négative renseignant un alignement gauche,
- `s` est une chaîne de caractères qui n'est pas obligatoire et qui permet de contrôler la mise en forme de la donnée à afficher.

Le langage C# propose aux programmeurs des formats pour les nombres et les dates (heures). De plus, il est possible, par un certain nombre d'options, de personnaliser certains formatages ayant trait aux nombres.

Nous allons maintenant passer ces différentes chaînes de caractères permettant de mettre en forme nos données à afficher :

- formatage concernant les nombres
 - `G` est le formatage standard de nombres,
 - `F` immédiatement suivi d'un nombre permet d'indiquer les décimales à afficher,
 - `N` s'applique aux grands nombres en groupant les chiffres par 3,
 - `E` permet la notation scientifique,
 - `D` s'applique aux entiers en les complétant, si nécessaire, à gauche par 0,
 - `C` permet un formatage financier (ou monétaire),
 - `P` est utilisé pour afficher des pourcentages,
 - `X` s'applique aux entiers et effectue l'affichage hexadécimal,

- formatage, dépendant des paramètres *régionaux* (France (Belgique)), concernant les dates et heures
 - Y fournit le mois (mot) et l'année,
 - M fournit le jour (nombre) et le mois (mot),
 - D fournit le jour (mot puis nombre), le mois (nom) et l'année,
 - d fournit la date sous le format *jj/mm/aaaa*,
 - T fournit l'heure sous le format *hh:mm:ss*,
 - t fournit l'heure sous le format *hh:mm*,
 - F fournit la date sous le format D et l'heure sous le format *hh:mm:ss*,
 - f fournit la date sous le format D et l'heure sous le format *hh:mm*,
 - G fournit la date et l'heure sous le format *jj/mm/aaaa hh:mm:ss*,
 - g fournit la date et l'heure sous le format *jj/mm/aaaa hh:mm*,
 - U est identique au format D mais utilise l'heure date universelle (UTC).

Nous pouvons, pour illustrer ces différents formats, examiner le code source suivant

```
using System;

class MonProg
{
    static void Main()
    {
        DateTime maintenant = DateTime.Now;
        Console.WriteLine("Formatage G : {0,10:G} et {1,10:G}", 12345, 12345.6);
        Console.WriteLine("Formatage F : {0,10:F2} et {1,10:F2}", 12345, 12345.6);
        Console.WriteLine("Formatage N : {0,10:N} et {1,10:N}", 12345, 12345.6);
        Console.WriteLine("Formatage E : {0,10:E} et {1,10:E}", 12345, 12345.6);
        Console.WriteLine("Formatage D : {0,10:D}", 12345);
        Console.WriteLine("Formatage C : {0,10:C} et {1,10:C}", 12345, 12345.6);
        Console.WriteLine("Formatage P : {0,10:P} et {1,10:P}", 12345, 12345.6);
        Console.WriteLine("Formatage X : {0,10:X}", 12345);
        Console.WriteLine("Formatage Y : {0:Y}", maintenant);
        Console.WriteLine("Formatage M : {0:M}", maintenant);
        Console.WriteLine("Formatage D : {0:D}\nFormatage d : {0:d}", maintenant);
        Console.WriteLine("Formatage T : {0:T}\nFormatage t : {0:t}", maintenant);
        Console.WriteLine("Formatage F : {0:F}\nFormatage f : {0:f}", maintenant);
        Console.WriteLine("Formatage G : {0:G}\nFormatage g : {0:g}", maintenant);
        Console.WriteLine("Formatage U : {0:U}", maintenant);
        Console.ReadLine();
    }
}
```

A l'exécution, le résultat devrait ressembler à

```
Formatage G :      12345 et      12345.6
Formatage F :    12345.00 et    12345.60
Formatage N :    12.345.00 et    12.345.60
Formatage E : 1.234500E+004 et 1.234560E+004
Formatage D :      12345
Formatage C : 12.345.00 ? et 12.345.60 ?
Formatage P : 1.234.500.00 % et 1.234.560.00 %
Formatage X :      3039
Formatage Y : juillet 2005
Formatage M : 25 juillet
Formatage D : lundi 25 juillet 2005
Formatage d : 25/07/2005
Formatage T : 10:03:46
Formatage t : 10:03
Formatage F : lundi 25 juillet 2005 10:03:46
Formatage f : lundi 25 juillet 2005 10:03
Formatage G : 25/07/2005 10:03:46
Formatage g : 25/07/2005 10:03
Formatage U : lundi 25 juillet 2005 8:03:46
```

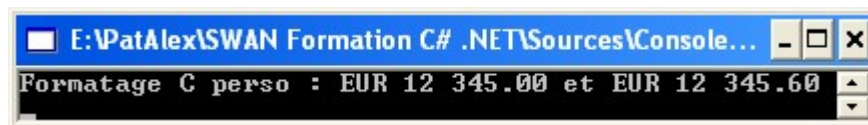
Il est également possible de personnaliser le formatage des données. Le sujet est relativement vaste et le lecteur intéressé pourra consulter l'aide sur les sujets `NumberFormatInfo` et `DateTimeFormatInfo`.

A titre d'illustration, nous pouvons considérer le source¹¹ suivant

```
using System;
using System.Globalization;

class MonProg
{
    static void Main()
    {
        NumberFormatInfo nfi = new NumberFormatInfo();
        nfi.CurrencyDecimalSeparator = ".";
        nfi.CurrencyGroupSeparator = " ";
        nfi.CurrencySymbol = "EUR ";
        Console.Write(String.Format(nfi, "Formatage C perso : {0,10:C}", 12345));
        Console.WriteLine(String.Format(nfi, " et {0,10:C}", 12345.6));
        Console.ReadLine();
    }
}
```

et obtenir, à l'exécution, un résultat semblable à



3.2.4. Un peu de fun dans la présentation

Par rapport à la version 1 de C#, nous avons la possibilité de personnaliser l'affichage en mode console. Nous n'allons pas nous appesantir sur ce sujet mais nous souhaitons signaler qu'un peu de couleur, de mise en forme voir même d'ambiance sonore sont actuellement rendus possible dans cette présentation austère, d'un *autre âge*, qu'est le mode console.

Ainsi, pour effacer un écran de type console, nous disposons de l'instruction

```
Console.Clear();
```

La couleur de fond se trouve modifiée par la propriété `BackgroundColor` sous la forme

```
Console.BackgroundColor = ConsoleColor.Blue;
```

où `ConsoleColor` permet d'accéder à une énumération de couleurs disponibles.

La couleur d'affichage se modifie par l'intermédiaire de la propriété `ForegroundColor` comme suit

```
Console.ForegroundColor = ConsoleColor.Yellow;
```

Nous pouvons également positionner le curseur où bon nous semble à l'intérieur de la fenêtre par l'intermédiaire de `Console.SetCursorPosition(num_colonne, num_ligne)`.

Signalons enfin que nous pouvons également personnaliser la fenêtre elle-même.

La zone de titre peut être personnalisée par l'intermédiaire de la propriété `Title` comme dans l'exemple ci dessous

```
Console.Title = "Personnalisons la console";
```

¹¹ Cet exemple nous permet de constater que les chaînes de formatage de données ne se limitent pas à la fonction `WriteLine()` et sa petite sœur `Write()`.

Ainsi, il est possible de spécifier le nombre de lignes et de colonnes souhaitées¹² pour l'affichage. Il nous suffit, pour travailler sur une fenêtre de 10 lignes sur 40 colonnes, de faire appel à l'instruction

```
Console.SetWindowSize(40, 10);
```

Rassemblant toutes ces instructions, nous pouvons rédiger le code suivant

```
using System;

class MonProg
{
    static void Main()
    {
        Console.SetWindowSize(40, 6);
        Console.Title = "Personnalisons la console";
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.Clear();
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.SetCursorPosition(16, 2);
        Console.WriteLine("I.S.E.T");
        Console.ForegroundColor = ConsoleColor.Red;
        Console.SetCursorPosition(4, 4);
        Console.WriteLine("Haute Ecole de la Ville de Liège");
        Console.ReadLine();
    }
}
```

pour obtenir le résultat



Cette même version 2 permet également aux mélomanes et aux autres de s'exprimer. La méthode `Console.Beep()` sans argument nous permet de faire *couiner* l'ordinateur. Néanmoins, cette même fonction existe également avec deux paramètres entiers qui renseignent respectivement

- la fréquence, en hertz, selon un spectre de valeurs comprises entre 37 et 32767,
- la durée, en millisecondes.

Pour les mélomanes ou ceux qui tendent à le devenir, nous pouvons trouver dans l'aide de Visual Studio l'énumération suivante qui permet de définir de manière plus manipulable les notes dans une octave, en ce compris le silence.

```
protected enum Note
{
    Silence=0, Sol#_196, La=220, La#=233, Si=247, Do=262, Do#=277, Re=294, Re#=311,
    Mi=330, Fa=349, Fa#=370, Sol=392, Sol#=415
};
```

¹² Les dimensions maximales disponibles sont renseignées par `Console.LargestWindowWidth` et `Console.LargestWindowHeight`.

Si nous souhaitons étendre quelque peu notre spectre, nous pouvons fouiller sur internet et découvrir, au détour d'une page¹³, le tableau suivant qui donne la fréquence d'un nombre un peu plus imposant de notes,

Octave	0	1	2	3	4	5	6	7
Note								
Do	32,70	65,41	130,81	261,63	523,25	1046,50	2093,00	4186,01
Do#	34,65	69,30	138,59	277,18	554,37	1108,73	2217,46	4434,92
Ré	36,71	73,42	146,83	293,66	587,33	1174,66	2349,32	4698,64
Ré#	38,89	77,78	155,56	311,13	622,25	1244,51	2489,02	4978,03
Mi	41,20	82,41	164,81	329,63	659,26	1318,51	2637,02	5274,04
Fa	43,65	87,31	174,61	349,23	698,46	1396,91	2793,83	5587,65
Fa#	46,25	92,50	185,00	369,99	739,99	1479,98	2959,96	5919,91
Sol	49,00	98,00	196,00	392,00	783,99	1567,98	3135,96	6271,93
Sol#	51,91	103,83	207,65	415,30	830,61	1661,22	3322,44	6644,88
La	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00	7040,00
La#	58,27	116,54	233,08	466,16	932,33	1864,66	3729,31	7458,62
Si	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07	7902,13

Notre enthousiasme doit toutefois être tempéré car la méthode `Console.Beep()` avec laquelle nous venons de faire connaissance se limite à des valeurs entières comprises entre 37 et 32767. Dès lors, nous transformons notre code précédent en

```
protected enum Frequence
{
    Silence=0,
    Do1=65, Do_1=69, Re1=73, Re_1=78, Mi1=82, Fa1=87, Fa_1=92,
    Sol1=98, Sol_1=104, La1=110, La_1=117, Si1=123,
    Do2=131, Do_2=139, Re2=147, Re_2=156, Mi2=165, Fa2=175, Fa_2=185,
    Sol2=196, Sol_2=208, La2=220, La_2=233, Si2=247,
    Do3=262, Do_3=277, Re3=294, Re_3=311, Mi3=330, Fa3=349, Fa_3=370,
    Sol3=392, Sol_3=415, La3=440, La_3=466, Si3=494,
    Do4=523, Do_4=554, Re4=587, Re_4=622, Mi4=659, Fa4=698, Fa_4=740,
    Sol4=784, Sol_4=831, La4=880, La_4=932, Si4=988,
    Do5=1046, Do_5=1109, Re5=1175, Re_5=1245, Mi5=1319, Fa5=1397, Fa_5=1480,
    Sol5=1568, Sol_5=1661, La5=1760, La_5=1865, Si5=1976,
    Do6=2093, Do_6=2217, Re6=2349, Re_6=2489, Mi6=2637, Fa6=2794, Fa_6=2960,
    Sol6=3136, Sol_6=3322, La6=3520, La_6=3729, Si6=3951,
    Do7=4186, Do_7=4435, Re7=4699, Re_7=4978, Mi7=5274, Fa7=5588, Fa_7=5920,
    Sol7=6272, Sol_7=6645, La7=7040, La_7=7459, Si7=7902,
};
```

Vient ensuite la durée :

```
protected enum Duree
{
    Ronde=1600, Blanche=Ronde/2, Noire=Blanche/2, Croche=Noire/2, DoubleCroche=Croche/2;
};
```

Même si cela dépasse le cadre de cette introduction, nous pouvons alors définir une classe destinée à gérer les notes. Nous écrivons

```
protected class Note
{
    Frequence ValFrequence;
    Duree ValDuree;
    public Note(Frequence frequence, Duree duree)
    {
        ValFrequence = frequence;
        ValDuree = duree;
    }
    public Frequence FrequenceNote { get { return ValFrequence; } }
    public Duree DureeNote { get { return ValDuree; } }
}
```

13 Par exemple, http://fr.wikipedia.org/wiki/Note_de_musique.

Sans entrer dans les détails, nous pouvons néanmoins commenter ce qui précède. Ainsi,

- les données propres à la classe `Note` sont `ValFrequence` et `ValDuree` qui trouvent leurs valeurs dans les énumérations `Frequence` et `Duree`,
- la méthode `Note` permet d'initialiser les données membres de la classe selon les deux paramètres transmis,
- les deux dernières entités sont des propriétés et donnent accès, en lecture, aux données membres de la classe.

Nous pouvons maintenant passer à la méthode qui permet de jouer un tableau d'objets de type `Note` passé en argument en décomposant chaque élément de ce tableau selon sa `Frequence` et sa `Duree`.

```
protected static void Play(Note[] Melodie)
{
    foreach (Note n in Melodie)
    {
        if (n.FrequenceNote == Frequence.Silence)
            Thread.Sleep((int)n.DureeNote);
        else
            Console.Beep((int)n.FrequenceNote, (int)n.DureeNote);
    }
}
```

La méthode `Main()` se résume alors à l'enregistrement du morceau que nous souhaitons faire jouer à l'ordinateur suivi de l'appel de la méthode `Play()` que nous venons de définir en lui transmettant le morceau de musique.

```
static void Main(string[] args)
{
    Note[] SotW = {
        new Note(Frequence.Silence, Duree.Croche),
        new Note(Frequence.Mi3, Duree.Blanche),
        new Note(Frequence.Sol3, Duree.Blanche),
        new Note(Frequence.La3, Duree.Blanche),
        new Note(Frequence.Silence, Duree.Croche),
        new Note(Frequence.Mi3, Duree.Blanche),
        new Note(Frequence.Sol_3, Duree.Blanche),
        new Note(Frequence.La_3, Duree.Noire),
        new Note(Frequence.La3, Duree.Noire),
        new Note(Frequence.Silence, Duree.Croche),
        new Note(Frequence.Mi3, Duree.Blanche),
        new Note(Frequence.Sol3, Duree.Blanche),
        new Note(Frequence.La3, Duree.Blanche),
        new Note(Frequence.Silence, Duree.Croche),
        new Note(Frequence.Sol3, Duree.Blanche),
        new Note(Frequence.Mi3, Duree.Blanche)
    };

    Play(SotW);
}
```

3.3. Les commentaires

Trois formes de commentaires sont possibles. Nous retrouvons les commentaires propres aux ancêtres du langage C# tels que C++ ou Java.

- `//` le reste de la ligne est ignoré par le compilateur,
- `/* ... */` tout ce qui est compris entre les symboles `/*` et `*/` est ignoré par le compilateur.

Nous en ajoutons un troisième qui sert à la documentation automatique du programme.

- `///` le reste de la ligne est ignoré par le compilateur mais sera pris en charge lors de la génération automatique de la documentation.

Les balises XML proposées par défaut pour récupérer automatiquement la documentation du code source sont les suivantes :

- `<summary>...</summary>`
encadre une brève description,
- `<remarks>...</remarks>`
encadre une description détaillée pouvant contenir des paragraphes imbriqués, des listes et d'autres types de balises,
- `<para>...</para>`
ajoute un paragraphe au sein d'une description, cette balise ne pouvant exister que si elle fait partie d'une autre balise,
- `<param name="nom">...</param>`
décrit un paramètre de méthode,
- `<returns>...</returns>`
documente la valeur renvoyée par la méthode,
- `<c>...</c>`
intègre un morceau de code, cette balise ne pouvant exister que si elle fait partie d'une autre balise,
- `<paramref name="nom"/>`
permet de lier l'apparition du paramètre dans un texte à sa définition, cette balise ne pouvant exister que si elle fait partie d'une autre balise,
- `<see cref="membre"/>`
permet de créer une référence à une entité définie dans .NET ou par nos soins, cette balise ne pouvant exister que si elle fait partie d'une autre balise,
- `<exception cref="membre"/>`
permet de créer une référence à une exception documentée dans .NET, cette balise ne pouvant exister que si elle fait partie d'une autre balise,
- `<code>...</code>`
intègre un morceau de code (plus conséquent que la balise `<c>`), associée à la balise `<example>`,
- `<example>...</example>`
permet d'illustrer des morceaux de code.

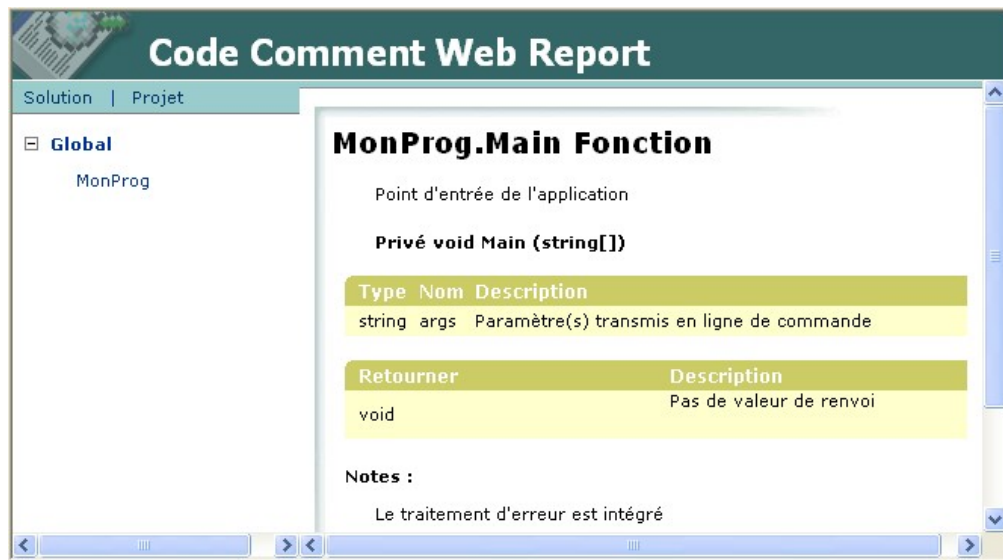
Comme illustration, nous pouvons reprendre le source commenté suivant

```

• using System;
•
• class MonProg
• {
•     /// <summary>
•     /// Point d'entrée de l'application
•     /// </summary>
•     /// <param name="args">Paramètre(s) transmis en ligne de commande</param>
•     /// <returns>Pas de valeur de renvoi</returns>
•     /// <remarks>Le traitement d'erreur est intégré</remarks>
•     static void Main()
•     {
•         System.Console.WriteLine("Hello World !");
•         System.Console.ReadLine();
•     }
• }

```

Lors de la génération de la documentation, il nous est possible d'admirer



Signalons, pour terminer, que la documentation réalisée est un fichier html utilisant des balises xml. Par conséquent, nous pouvons également utiliser des balises xml standard. Prenons l'exemple d'une liste hiérarchisée que nous ajoutons dans un bloc de commentaire de type `<remarks>`.

Nous pouvons écrire

```

using System;

class MonProg
{
    /// <summary>
    /// Point d'entrée de l'application
    /// </summary>
    /// <remarks>
    /// <list type="bullet">
    /// <item><b>GRAS</b></item>
    /// <item><i>ITALIQUE</i></item>
    /// <item><u>SOULIGNE</u></item>
    /// </list>
    /// </remarks>
    static void Main()
    {
        System.Console.WriteLine("Hello World !");
        System.Console.ReadLine();
    }
}

```

et pourquoi pas des balises de type `` pour ouvrir une fenêtre avec l'adresse renseignée.

4. Les types

Nous allons maintenant nous pencher sur tout ce qui concerne les variables et disposerons ainsi, à la fin du chapitre, de ressources nécessaires au stockage temporaire des grandeurs.

4.1. Les identificateurs en C#

Les variables se déclarent de la même manière qu'en C. Ainsi, les lettres minuscules et majuscules sont à nouveau différenciées. Néanmoins, il existe deux différences notables :

- les lettres accentuées sont maintenant permises dans les noms de variables,
- il n'y a pas de limite quant au nombre de caractères utilisés pour baptiser une variable.

Pour les caractères forts qui n'acceptent pas bien les règles imposées par le compilateur, il est possible d'empiéter sur son domaine en utilisant de mots réservés. Evidemment, le compilateur ne laisse pas partir ses prérogatives sans contre-partie et il faut préfixer le nom réservé par le symbole @. Ainsi, nous pourrions envisager d'utiliser les variables @switch, @if, @for, ...

Pour information, les mots réservés sont les mots clés suivants, sur lesquels nous reviendrons par la suite, au fur et à mesure de la découverte du langage,

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	var	virtual	void	volatile	while

Comme annoncé, il n'y a pas de variables globales. Néanmoins, il existe une possibilité de contourner cette restriction. En effet, déclarer un membre de la classe MonProg, dans l'exemple présenté auparavant, et l'affubler du *qualificatif* static le rend accessible à la classe entière.

4.2. Les types de données

Comme dans tout langage de programmation qui se respecte, nous pouvons distinguer différents types de données. Nous retrouvons évidemment un air de parenté avec ce que nous connaissons en C mais nous devons toutefois signaler qu'un des gros avantages de l'environnement .NET est d'avoir harmonisé les différents types de données dans les différents langages supportés.

Ces types de données font partie du *Système de types communs* (CTS : Common Type System) et les langages qui souhaitent s'intégrer dans la plate-forme .NET doivent essayer de s'y plier pour conserver la compatibilité avec les autres.

Les types de ce CTS se subdivisent en deux parties :

- les variables de type valeur
- contiennent directement les données,
- disposent de leur propre jeu de données,
- ne sont pas affectées par les opérations effectuées sur d'autres variables,
- les variables de type référence
- stockent des références aux données et non les données elles-mêmes,
- peuvent partager les mêmes données,
- sont affectées par les opérations effectuées par d'autres variables (le contenu des données référencées),
- suppriment les données qu'elles renseignent par une affectation à `null`.

Le lecteur attentif reconnaîtra la dualité entre les variables classiques et les variables de type pointeurs du langage C si ce n'est qu'il s'agit maintenant de variables *référence*¹⁴.

Nous commençons par les types valeurs, à savoir

- les types entiers
 - le type `byte` est codé sur un octet et prend n'importe quelle valeur comprise entre 0 et 255,
 - le type `sbyte` est également codé sur un octet mais, pour sa part, prend n'importe quelle valeur comprise entre -128 et 127,
 - le type `short` est codé sur 2 octets et peut prendre toutes les valeurs entières comprises entre -32768 (-2^{15}) et 32767 ($2^{15}-1$),
 - le type `ushort` est similaire à `short` mais ne considère que les entiers positifs et, par conséquent, peut prendre les valeurs entre 0 et 65535¹⁵,
 - le type `int` double la capacité de stockage (4 octets) et permet de manipuler des entiers compris entre -2147483648 (-2^{31}) et 2147483647 ($2^{31}-1$),
 - le type `uint` est semblable à `int` mais ne concerne que les entiers positifs compris entre 0 et 4294967293,
 - le type `long` double à nouveau la capacité de stockage (8 octets) et concerne des entiers compris entre $-9,2 \times 10^{18}$ et $9,2 \times 10^{18}$,
 - le type `ulong` reprend les caractéristiques de `long` mais se limite aux entiers positifs inférieurs à $1,8 \times 10^{19}$.

¹⁴ Sans s'étendre sur le sujet, signalons qu'il s'agit de variables de type pointeurs où l'écriture propre à cet environnement a été allégée. Une fois déclarée comme étant une variable référence (stockant l'adresse d'une entité plutôt que ses valeurs), la manipulation dans le code source est tout à fait similaire à l'utilisation d'une variable classique. La lecture en est facilitée quoique il ne faut pas perdre de vue ce que l'on fait réellement ...

¹⁵ Les types entiers `unsigned` sont pris en considération par le seul langage C# et, tant que le programmeur travaille en C#, il n'y a pas de problème.
Par contre, si des échanges doivent avoir lieu avec des modules rédigés en un autre langage, il y aura des adaptations à réaliser.

- le type booléen (logique) peut contenir les valeurs `true` et `false` pour lesquels il ne faut plus chercher d'équivalent entier et sera déclaré par le mot réservé `bool`,
- les types réels
 - le type `float` permet de manipuler des nombres réels en simple précision (7 décimales), codés sur 4 octets et pour lesquels les plus petites et plus grandes valeurs (en valeur absolue) sont $1,4 \times 10^{-45}$ et $3,4 \times 10^{38}$,
 - le type `double` rend disponible des nombres réels en double précision (15 décimales), codés sur 8 octets et pour lesquels les plus petites et plus grandes valeurs (en valeur absolue) sont $4,9 \times 10^{-324}$ et $1,8 \times 10^{308}$,
 - le type `decimal` permet à l'utilisateur de travailler avec des nombres réels sous la forme de nombres entiers multipliés par une puissance de 10 (limité à 8×10^{28}).
- le type `char`, contrairement au C est codé sur 2 octets pour permettre l'utilisation du système Unicode prenant notamment en charge d'autres alphabets,
- les chaînes de caractères sont des objets issus de la classe `string` pour lesquels, notamment, il n'est plus question de préciser à l'avance la taille maximum souhaitée,

Signalons, pour terminer, que les types présentés ci-dessus peuvent en fait être considérés comme des objets en provenance de classes. Ces types sont également qualifiés d'*intégrés*.

Nous avons le tableau de correspondances suivant

Mot clé	Alias System
<code>sbyte</code>	<code>System.SByte</code>
<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>bool</code>	<code>System.Boolean</code>
<code>float</code>	<code>System.Single</code>
<code>double</code>	<code>System.Double</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>char</code>	<code>System.Char</code>
<code>string</code>	<code>System.String</code>

Par conséquent, il est possible de remplacer la déclaration

```
int i;
```

par la suivante, équivalente,

```
System.Int32 i;
```

4.3. La manipulation des variables et des constantes

4.3.1. Les variables

Classiquement, il nous faut spécifier le type de la variable puis renseigner le nom de baptême. Une initialisation peut être effectuée lors de la déclaration et elle ne se limite pas à affecter une valeur constante. Nous pouvons utiliser un calcul ou même effectuer un appel à une fonction.

Ainsi, les déclarations suivantes sont correctes

```
bool drapeau;  
int i, j;  
double pi=Math.PI;  
double r=2.75;  
double surf=pi*Math.Pow(r, 2);
```

Nous noterons également trois différences avec notre étude antérieure du langage de programmation C.

Premièrement, les variables pour lesquelles aucune valeur n'a été affectée ne peuvent être utilisées dans le membre de droite d'une affectation, le refus se faisant à la compilation. Cela permet d'éviter une assignation par une variable dont la valeur est inconnue.

Deuxièmement, la reconnaissance des valeurs assignées aux variables se fait selon certaines règles. Un nombre entier est assimilé à un entier de type `int` tandis que pour les réels, il s'agit du type `double`.

Troisièmement, les conversions de type sont moins automatiques ou, de manière équivalente, nous pouvons dire que C# est quelque peu plus regardant. Cela peut entraîner des erreurs surprenantes, essentiellement quant au type réel. Par exemple, l'instruction

```
float pi = 3.14159;
```

pose un problème de syntaxe car 3.14159 est, par défaut, de type `double` et, par conséquent, il ne peut être converti en `float`, de précision moindre.

Deux possibilités s'offrent au programmeur pour adapter le type de la donnée numérique au type de la variable qui la reçoit, le transtypage (*casting*) traditionnel

```
float pi = (float)3.14159;
```

ou la mise en place d'un suffixe de conversion comme

```
float pi = 3.14159f;
```

Signalons pour terminer que pour travailler avec des données numériques constantes en mode

- `long`, le suffixe de conversion est `l` ou `L`,
- `decimal`, le suffixe de conversion est `m` ou `M`,
- `float`, le suffixe de conversion est `f` ou `F`,
- `double`, le suffixe de conversion est `d` ou `D`.

4.3.2. Les constantes

Les constantes se définissent par l'intermédiaire du mot clé `const` et, par conséquent, nous pouvons oublier l'instruction `#define`, utilisée à d'autres fins.

En ce qui concerne l'utilisation du mot réservé `const`, il suffit de le faire suivre par le type de données concernée puis du nom de la constante et, enfin, de la valeur attribuée.

Ainsi, nous pouvons écrire

```
const int nb = 25;
```

et nous pouvons noter qu'il est possible d'utiliser une constante définie au préalable pour définir la valeur associée à une nouvelle constante.

Par exemple, en tenant compte de qui précède, nous pouvons écrire

```
const int jeux = 3*nb;
```

Nous venons de terminer les types intégrés de données, c'est-à-dire ceux fournis par le langage. A côté de ces types, nous avons les types définis par l'utilisateur.

4.4. Le type énuméré

Un entier de type `short` peut être considéré comme un type énuméré car les variables de ce type peuvent prendre leur valeur dans l'ensemble $\{-128, -127, \dots, 126, 127\}$.

C# laisse également la possibilité à l'utilisateur de définir ses propres types énumérés par l'intermédiaire de la commande `enum`. Par exemple, nous pourrions écrire

```
enum joursemaine {lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche};
```

En nous basant sur le programme habituel, nous pouvons intégrer ce nouveau type dans l'application et écrire

```
using System;

class MonProg
{
    enum joursemaine {lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche};

    static void Main()
    {
        joursemaine queljour;
        // queljour est du type enumere joursemaine
        queljour = joursemaine.mercredi;
        // Initialisation de la variable enumeree selon la liste définie dans joursemaine
        Console.WriteLine("Résultat : {0}",queljour);
        // Affichage de mercredi
        Console.ReadLine();
    }
}
```

Comme on pourrait s'en douter, un type énuméré met en bijection les éléments définis avec une numérotation. Par défaut, le premier élément porte le numéro 0, le second le numéro 1, ... Il est possible de définir sa propre numérotation¹⁶ et de se déplacer au sein des valeurs possibles d'un tel type par des opérations sur les numéros. Par exemple, l'instruction de casting

```
queljour = (joursemaine) 2;
```

permet d'assigner, de manière équivalente, à la variable `queljour` la valeur `mercredi`.

Toutefois, il est possible de redéfinir le point de départ de cette numérotation :

```
enum joursemaine {lundi=1,mardi,mercredi,jeudi,vendredi,samedi,dimanche};
```

¹⁶ Le numéro 0 est conseillé car il permet de réaliser des initialisations par défaut.

4.5. Les structures

La définition de base d'une structure est identique à celle rencontrée en C. Deux différences apparaissent néanmoins.

- le symbole ; de fin de définition est superflu mais reste accepté,
- les données membres d'une structure doivent être qualifiées de `public` pour être accessibles de l'extérieur.

En fait, les structures et les classes sont relativement proches l'une de l'autre. Nous présentons ici brièvement les distinctions existantes que nous développerons plus en profondeur lors de l'étude des classes. Nous relevons les différences suivantes

- les structures sont de type *valeur* (les classes sont de type *référence*, c'est-à-dire une *adresse* déguisée),
- l'héritage est inexistant pour les structures contrairement aux classes (toutes dérivent de la classe `Object`),
- les champs d'une structure ne peuvent être initialisés dans leur déclaration au sein de la structure,
- une structure peut contenir 0, 1 ou plusieurs constructeurs (méthode particulière appelée lors de la création de la variable) mais pas de constructeur sans argument (constructeur par défaut).

Une autre particularité des structures concerne l'utilisation de la commande `new`. En considérant la définition

```
• struct etud
• {public string nom;
•   public int cote;
• }
```

la déclaration de la variable `eleve` par l'instruction

```
• etud eleve;
```

n'entraîne aucune initialisation des données membres contrairement à l'instruction suivante qui initialise les données membres à 0¹⁷.

```
• etud eleve = new etud();
```

Le remplissage des données concernant cette variable `eleve` se fait de manière *classique*

```
• eleve.nom = "Winch";
• eleve.cote = 19;
```

¹⁷ L'initialisation à 0 se fait au sens large car, pour les données membres de type `string`, l'initialisation se fait selon la chaîne vide.

4.6. Les tableaux

Les tableaux représentent une des améliorations notables en ce qui concerne le travail des développeurs.

Commençons par le début, à savoir la déclaration. Pour pouvoir utiliser un tableau composé d'un certain nombre d'entiers, nous pouvons simplement écrire

```
int[] vecteur;
```

Nous pouvons alors mettre en place une référence au tableau `vecteur` que nous devons encore définir et utiliser. Si ce dernier doit contenir 5 entiers, nous pouvons alors, par la suite, utiliser l'instruction

```
vecteur = new int[5];
```

Il est possible de *condenser* les deux écritures précédentes sous la forme

```
int[] vecteur = new int[5];
```

Nous pouvons également initialiser le tableau quant au contenu des ses cellules en spécifiant les valeurs que nous voulons voir apparaître par l'écriture

```
vecteur = new int[] {0,5,10,15,20};
```

Cette opération peut également être effectuée dans le même temps que la déclaration sous la forme

```
int[] vecteur = {0,5,10,15,20};
```

Enfin, une nouvelle définition peut se faire en cours de programme. C'est ainsi que les deux instructions suivantes ne sont nullement incompatibles

```
int[] vecteur = {0,5,10,15,20};  
vecteur = new int[] {0,10,20,30,40,50,60,70,80,90,100};
```

La numérotation des cellules se fait comme en C, en commençant par 0, l'indice de la dernière cellule étant alors $n-1$ pour un tableau de dimension n .

Point non négligeable pour les programmeurs, en particulier ceux qui ne sont pas prudents avec les indices de tableaux, il n'est pas possible, en C#, de dépasser les limites définies pour un tableau. Par exemple, partant de la déclaration

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

les instructions

```
vecteur[-2] = 18;
```

ou

```
Console.WriteLine(vecteur[15]);
```

conduisent à une erreur en cours d'exécution¹⁸. Nous évitons ainsi des *écrasements* indésirables de contenu d'emplacements en mémoire.

¹⁸ Le programmeur n'a pas gagné le paradis car ce type de dépassement de borne n'est (encore) pas signalé à la compilation même si l'indice est une constante.

Pour déclarer un tableau à plusieurs dimension comme, par exemple, une matrice, nous pouvons écrire, selon nos besoins,

```
int[,] matrice1;  
int[,] matrice = new int[3,3];  
int[,] matrice = {{1,2,3},{4,5,6},{7,8,9}};
```

Dans le cadre de ces tableaux multidimensionnels, nous pouvons relever la méthode `GetLength()` qui permet de récupérer le rang (nombre maximum d'éléments) d'une dimension. Par la même occasion, nous présentons ci après le fonctionnement des propriétés `Length` et `Rank` Par exemple, pour la déclaration

```
int[,] matrice = new int[5,3];
```

nous avons

- `matrice.Length` vaut 15 (5×3),
- `matrice.Rank` vaut 5,
- `matrice.GetLength(0)` vaut 5,
- `matrice.GetLength(1)` vaut 3,

Pour conclure ce paragraphe, rappelons que

- un tableau est un *groupement* de données de même type,
- un tableau ne peut se redimensionner lorsqu'il est rempli,
- l'accès aux éléments d'un tableau ne peut être en lecture seule,
- de par la structure statique et organisée, les manipulations sont rapides mais rigides.

Signalons tout d'abord qu'il est possible de lever la première contrainte. Puisque C# axe tout sur les classes, nous pouvons déclarer un tableau d'objets (basiques) et y placer des entiers, réels, string, ... Par exemple, l'écriture suivante est permise :

```
object[] vecteur = {5,"Marcel",3.14159,true};
```

De manière plus conventionnelle, un nouveau type apparaît : la collection. Cette dernière permet de contourner l'aspect peu souple des tableaux mais les performances s'en ressentent.

Ainsi, les collections permettent de

- considérer des *tableaux* composés de données de types différents,
- de redimensionner dynamiquement le *tableau*,
- l'accès aux éléments peut se faire en lecture seule.

Les scrupules philosophiques concernant ce type identique, en apparence, au tableau tel qu'on le connaît dans d'autres langages peuvent être ignorés car les collections sont dérivées de la classe `Array` ou, plus exactement, `ArrayList`. Nous pouvons voir dans cette classe l'implémentation des listes linéaires (non homogènes). Nous pouvons écrire

```
ArrayList flex = new ArrayList();  
flex.Add(4);  
flex.Add("Hello ");  
flex.Add("World ");  
flex.Add("!");  
flex.Add(true);
```

Toujours dans le cadre des particularités, nous pouvons également déclarer des tableaux qualifiés de *déchiquetés*, c'est-à-dire des tableaux dont les nombres d'éléments varient au sein même de la structure. Par exemple, nous pourrions avoir besoin de traiter des matrices dont la première ligne est composée de 4 entiers, la deuxième ligne de 6 ... et, pour terminer, la troisième ligne composée de 2 entiers. Il suffit d'écrire

```
int[][] dent_de_scie = new int[3][];  
dent_de_scie[0] = new int[4];  
dent_de_scie[1] = new int[6];  
dent_de_scie[2] = new int[]{50,100};
```

Nous retrouvons ici, tacitement, l'organisation sous forme de pointeurs telle qu'elle est mise en place en C.

Les tableaux disposent de leurs propres méthodes. La plus particulière d'entre elles concernent la copie d'un tableau. En effet, les instructions

```
int[] original = {0,5,10,15,20};  
int[] copie = original;
```

font en sorte que `copie` référence les mêmes données que `original`. Si nous souhaitons réaliser une copie effective des données, la méthode `Clone()` est à notre disposition :

```
int[] original = {0,5,10,15,20};  
int[] copie = (int []) original.Clone();
```

5. Les instructions

Un programme se compose d'une suite d'instructions. Au moment de l'exécution, ces instructions sont exécutées les unes après les autres par la machine, dans l'ordre dans lequel elles se présentent. Nous allons maintenant voir de quelles instructions de base nous disposons en C#.

5.1. Les blocs d'instructions

Rien de neuf sous le soleil. Nous pouvons conserver nos habitudes en provenance du C puisque les blocs d'instructions sont renseignés par { et }.

Contrairement à ce qui se passe en C, la déclaration peut se faire en cours de programme et non en en-tête. Nous pouvons même aller plus loin dans cette déclaration déplacée en précisant que la déclaration est locale au bloc où elle est effectuée. Ainsi, nous pouvons écrire

```
{int i;  
//...  
}  
//...  
{int i;  
//...  
}
```

Parallèlement à ce qui se passe en C, il ne faut pas essayer de déclarer, au sein d'un sous-bloc, une variable de même nom que dans le bloc principal. Par conséquent, l'écriture suivante est rejetée à la compilation

```
{int i;  
//...  
    {int i;  
    //...  
    }  
//...  
}
```

5.2. Les initialisations et les affectations

Un des piliers de la programmation est la variable. Il n'y a pas guère de différence entre C# et ses prédécesseurs. Par conséquent, nous allons, dans ce qui suit, simplement effectuer un rappel de ce qui se passe en C en l'adaptant au contexte propre à C#.

5.2.1. Initialiser une variable

Il est possible d'attribuer une valeur à une variable dès sa déclaration. Pour ce faire, il suffit de faire suivre le nom de la variable par le signe = puis par la valeur souhaitée.

Par exemple, nous pourrions, dans le programme précédent, souhaiter initialiser la variable entière à 5 et celle de type réel à 3,14159... Nous aurions alors, en résumé,

```
using System;

class MonProg
{
    static void Main()
    {
        int i=5, j=10;
        float nbrePI=3.14159265358979323846264f;
        Console.WriteLine("Le nombre entier : {0} et {1}", i, j);
        Console.WriteLine("Le nombre réel : {0}", nbrePI);
        Console.ReadLine();
    }
}
```

Rappelons, au passage, le *casting* de la valeur renseignée pour `nbrePI` obligatoire car une telle valeur constante est considérée, par défaut, comme de type `double`. Cette conversion se fait en postfixant la valeur par `f`. Nous avons abordé ce sujet en 4.3.1.

5.2.2. Assigner des variables

La déclaration de variables n'est heureusement pas le seul endroit du programme où nous pouvons affecter une valeur à une variable. En fait, comme en C, à (presque) n'importe quel endroit, il est possible d'associer une valeur (conforme) à une variable et cette opération se nomme alors l'assignation. Le principe est identique à celui de l'initialisation. Il suffit de faire suivre le nom de la variable visée par le signe `=` puis par la valeur à mettre en place.

Par exemple, nous pouvons compléter le programme précédent selon

```
using System;

class MonProg
{
    static void Main()
    {
        int i=5;
        float nbrePI=3.14159265358979323846264f;
        Console.WriteLine("Initialisation");
        Console.WriteLine("Le nombre entier : {0}", i);
        Console.WriteLine("Le nombre réel : {0}", nbrePI);
        Console.WriteLine("Assignment");
        i=j=20;
        nbrePI=2.71828182846f;
        Console.WriteLine("Le nombre entier : {0}", i);
        Console.WriteLine("Le nombre réel : {0}", nbrePI);
        Console.ReadLine();
    }
}
```

La double assignation concernant les variables `i` et `j` est toujours d'application.

Nous pouvons maintenant passer aux calculs à proprement parler. Evidemment, il n'est pas question de révolution dans ce cadre et nous retrouvons les opérations rencontrées en C ou dans bien d'autres langages de programmation.

5.3. Des variables *atypiques*

Maintenant que nous avons abordé l'initialisation de variables, nous pouvons présenter une nouveauté apparue avec la troisième version du langage C# : utilisation de variables locales typées implicitement.

Cela signifie que le développeur n'est pas obligé de déclarer le type de la variable qu'il utilise, l'association se faisant alors par l'intermédiaire de l'initialisation effectuée. Le mot clé `var` sera utilisé pour annoncer une telle variable.

C'est ainsi que nous pouvons écrire

```
var s = @"c:\program files\";
```

pour déclarer la variable `s` qui sera finalement de type `string`.

Evidemment, le compilateur rejettera une telle déclaration si elle n'est pas accompagnée de son initialisation comme l'instruction suivante.

```
var s;
```

Nous pouvons associer à une variable locale typée implicitement tout type reconnu par le langage C#.

L'instruction permettant de déclarer et d'initialiser la variable `s` est bien entendu équivalente à celle rencontrée jusqu'alors

```
string s = @"c:\program files\";
```

La différence se marque par l'utilisation que nous pouvons faire de ce nouveau type de variable.

5.4. Opérations sur les variables

Nous passons maintenant à la mise en œuvre des différentes opérations arithmétiques.

5.4.1. Opérations arithmétiques standards

Evidemment, nous retrouvons les opérations arithmétiques¹⁹ de base :

- `+` effectue l'addition,
- `-` effectue la soustraction,
- `*` effectue la multiplication,
- `/` effectue la division.

¹⁹ Tout comme pour la machine à calculer, les opérateurs arithmétiques subissent un ordre de priorité. Néanmoins, les parenthèses sont ici aussi disponibles.

Nous pouvons néanmoins ajouter

- `%` calcule le reste de la division entre entiers.

ainsi que les raccourcis

- `+=` ajoute la valeur renseignée (à droite)
`i+=20;` équivaut à l'instruction `i=i+20;`
- `++` ajoute 1
`i++;` équivaut à l'instruction `i=i+1;` mais commence par transmettre la valeur de `i` avant d'ajouter 1,
`++i;` équivaut à l'instruction `i=i+1;` mais commence par incrémenter avant de transmettre la valeur de `i`,
- `-=` soustrait la valeur renseignée (à droite)
`i-=20;` équivaut à l'instruction `i=i-20;`
- `--` soustrait 1
`i--;` équivaut à l'instruction `i=i-1;` mais commence par transmettre la valeur de `i` avant de soustraire 1,
`--i;` équivaut à l'instruction `i=i-1;` mais commence par diminuer de 1 la valeur de `i` avant de la transmettre,
- `*` multiplie par la valeur renseignée (à droite)
`i*=20;` équivaut à l'instruction `i=i*20;`
- `/` divise par la valeur renseignée (à droite)
`i/=20;` équivaut à l'instruction `i=i/20;`
- `%` fournit le reste de la division par la valeur renseignée (à droite)
`i%=20;` équivaut à l'instruction `i=i%20;`

5.4.2. Opérateurs de comparaison

Nous devons également construire des expressions booléennes. Pour ce faire, nous commençons par construire une expression booléenne par l'intermédiaire des opérateurs de comparaison :

- `==` pour **est égal à**,
- `!=` pour **est différent de**,
- `<` pour **est strictement inférieur ou égal à**,
- `<` pour **est strictement inférieur à**,
- `<=` pour **est inférieur ou égal à**,
- `>` pour **est strictement supérieur à**,
- `>=` pour **est supérieur ou égal à**.
- `is` pour **est de type compatible avec**,

Seul le dernier opérateur est quelque peu nouveau. Nous pouvons reprendre l'exemple²⁰ fourni avec la documentation de Visual Studio .NET pour cerner son champ d'application :

```
using System;

class Classe1;
{
}
class Classe2;
{
}
class MonProg
{
    public static void Test(object obj)
    {
        Classe1 c1;
        Classe2 c2;
        if(obj is Classe1)
        {
            Console.WriteLine("Compatibilité avec Classe1");
            c1=(Classe1) obj;
            //...
        }
        else if(obj is Classe2)
        {
            Console.WriteLine("Compatibilité avec Classe2");
            c2=(Classe2) obj;
            //...
        }
        else Console.WriteLine("Incompatibilité totale");
    }
    public static void Main()
    {
        Classe1 a;
        Classe2 b;
        Test(a);
        Test(b);
        Test("Une chaîne de caractères");
    }
}
```

Ensuite, une fois que nous sommes en mesure d'écrire des expressions booléennes de base, nous pouvons les combiner à l'aide des opérateurs booléens :

- && pour la conjonction logique (et),
- || pour la disjonction logique (ou).

5.5. Le branchement conditionnel (simple)

Il s'agit ici d'aborder la structure conditionnelle

Si condition alors faire instruction₁ sinon faire instruction₂.

En ajoutant que la partie relative à la non-vérification de la condition est facultative et que, par conséquent, nous pouvons avoir

Si condition alors faire instruction.

En C#, de manière schématique, cela se traduit par

```
if (condition) instruction1;
else instruction2;
```

ou, en cas d'absence d'alternative,

```
if (condition) instruction;
```

²⁰ Cet exemple arrive un peu tôt puisque nous y découvrons l'instruction if-else ainsi que la définition et l'utilisation de la fonction Test() mais, comme c'est identique à ce que nous avons vu en C, cela ne devrait pas poser de problème.

Pour obtenir la valeur absolue d'un nombre, nous pouvons écrire

```
if (var>0) resultat = var;
else resultat = -var;
```

ou, de manière équivalente mais plus *succinte*,

```
resultat = (var>0) ? var : -var;
```

5.6. Le branchement conditionnel multiple

Le branchement multiple permet, en fonction du contenu d'une variable de contrôle, un aiguillage vers l'instruction correspondante. Nous avons, de manière générale,

Dans le cas où la variable de contrôle vaut

a, faire instruction_a,

b, faire instruction_b,

...

sinon, faire instruction_{défaut}.

En C#, la variable de contrôle est de type entier, `char`, `enum` ou `string` et la syntaxe est la suivante, basée sur la variable de contrôle et les valeurs valeur1, valeur2, ... qu'elle peut prendre,

```
switch (varctrl)
{
    case valeur1: instruction1;
        break;
    case valeur2: instruction2;
        break;
    //...
    default: instruction_défaut;
}
```

Par exemple, nous pouvons écrire

```
using System;

class MonProg
{
    enum mois { Janvier, Février, Mars, Avril, Mai, Juin, Juillet,
               Août, Septembre, Octobre, Novembre, Décembre };
    static void Main()
    {
        short nbjours;
        mois quelmois;
        // ...
        switch (quelmois)
        {
            case mois.Février: nbjours=28;
                                // Peut mieux faire ...
                                break;

            case mois.Avril :
            case mois.Juin :
            case mois.Septembre:
            case mois.Novembre : nbjours=30 ;
                                break;

            default: nbjours=31;
                    break;
        }
        //...
    }
}
```

Nous remarquons que cette instruction de branchement multiple entraîne l'apparition d'une instruction nouvelle, à savoir `break` qui signale la fin de l'instruction liée à la variable de contrôle.

Nous ajouterons que cette instruction `break` est devenue nécessaire lorsqu'un branchement comporte une instruction ou plus. Le compilateur bloque dans le cas contraire.

5.7. La boucle *statique*

L'instruction suivante permet d'exécuter une instruction un certain nombre de fois, déterminé à l'avance et se met, schématiquement, sous la forme

Répéter **n** fois **Faire instruction**.

ou, de manière équivalente, en utilisant une variable pour effectuer le décompte,

Pour une **variable** prenant les valeurs **1** à **n**, **faire instruction**.

En C#, nous pouvons écrire²¹

```
int cpt;
//...
for (int i=debut; i<=fin; i++) instruction;
```

où la variable `i` débute par la valeur `debut` et, tant qu'elle est inférieure à `fin`, exécute l'instruction en se voyant augmentée de 1 à chaque fin de boucle. Une fois la boucle exécutée, la variable `i` vaut `fin+1`.

Nous pouvons revenir au programme présenté en 3.1.3 qui permet de récupérer les arguments transmis en ligne de commande et le compléter de la manière suivante

```
using System;

class MonProg
{
    static void Main(string[] args)
    {
        for (int i=0; i<args.Length; i++)
        {
            Console.WriteLine("Argument {0} : {1}", 1+i, args[i]);
            Console.ReadLine();
        }
    }
}
```

Voici un exemple de fonctionnement

```
C:\WINDOWS\system32\cmd.exe - monprog "coucou les petits loups"
E:\PatAlex\SWAN Formation C# .NET\Sources\MonProg\bin\Debug>monprog coucou les p
etits loups
Arguments 1 : coucou
Arguments 2 : les
Arguments 3 : petits
Arguments 4 : loups

E:\PatAlex\SWAN Formation C# .NET\Sources\MonProg\bin\Debug>monprog "coucou les
petits loups"
Arguments 1 : coucou les petits loups
```

²¹ Notons que `debut` et `fin` sont placés dans l'instruction en lieu et place d'une valeur numérique, une variable ou une constante. Il ne s'agit donc là que des symboles uniquement utilisés pour la bonne compréhension de cette instruction.

5.8. La boucle *dynamique* (condition en entrée de boucle)

Le programmeur dispose également de boucles dont l'exécution est liée à une condition pouvant être modifiée en cours de programme et dont on ne connaît pas, a priori, le nombre de *passage*. La première boucle de ce type place la condition en entrée de la manière suivante :

Tant que **condition**, **faire instruction**.

En C#, il nous suffit de traduire et nous disposons de l'instruction

```
• while (condition) instruction; •
```

où nous constatons qu'il est tout à fait possible de ne pas exécuter `instruction` lorsque `condition` n'est pas vérifié en entrée de boucle tout comme le programme peut boucler indéfiniment si `condition` reste invariablement vrai, ce qui est nettement plus gênant.

Pour illustrer ce type de boucle dynamique, nous allons rédiger un programme qui permet de trouver le zéro d'un polynôme quelconque dans un intervalle donné. Notons que, pour être sûr que l'algorithme fonctionne, le polynôme doit être de signe différent aux deux extrémités de l'axe. Dès lors, nous utilisons une méthode dichotomique

- l'intervalle se rétrécit de moitié de manière à ce que le polynôme soit toujours de signe différent aux extrémités des intervalles construits,
- la méthode stoppe lorsque l'intervalle ainsi obtenu atteint une largeur inférieure à la précision souhaitée.

Voici le code source C# de cet exemple dont nous avons rencontré la version C lors des laboratoires de 1^{ère}.

```
• using System;
•
• class MonProg
• {static void Main()
• {float eps=1e-6f,a=0,b=5,c,pol;
•   while (b-a>eps)
•   {c=(a+b)/2;
•     pol=1+c-c*c+c*c*c/8;
•     if (pol*(1+a-a*a+a*a*a/8)>0) a=c;
•     else b=c;
•   }
•   Console.WriteLine("La racine vaut {0}",c);
•   Console.ReadLine();
• }
• }
```

5.9. La boucle *dynamique* (condition en sortie de boucle)

La seconde boucle de type dynamique place la condition en fin d'exécution de la manière suivante

Faire instruction tant que **condition**.

En C#, nous avons

```
• do instruction; while (condition); •
```

où nous constatons que `instruction` sera de toute manière exécuté au moins une fois et que, comme précédemment, le programme peut boucler indéfiniment si `condition` reste invariablement vrai, ce qui est nettement plus gênant.

L'exemple suivant permet d'encoder une suite (non vide) de nombres entiers et en calcule la moyenne et la variance. La boucle `do . . . while` permet de terminer la liste de nombres.

```

using System;

class MonProg
{
    static void Main()
    {
        int nb, cb=0;
        float som=0f, car=0f;
        string rep;
        do {
            Console.Write("Votre nombre : ");
            nb=Int32.Parse(Console.ReadLine());
            cb++; som+=nb; car+=nb*nb;
            Console.Write("Voulez-vous continuer (o/n) ?");
            rep=Console.ReadLine();
        }
        while (rep=="o" || rep=="O");
        Console.WriteLine("La moyenne vaut : {0}", som/cb);
        Console.WriteLine("La variance vaut : {0}", car/cb-som*som/(cb*cb));
        Console.ReadLine();
    }
}

```

5.10. L'instruction de parcours de collection

Vient s'ajouter aux boucles classiques une nouvelle qui permet de parcourir toute collection définie dans C#. Ces collections sont en fait des entités logicielles qui permettent de collecter d'autres entités logicielles tout comme une maison est une collection de pièces ou un tableau est une collection d'éléments.

Il s'agit donc d'une instruction qui peut se mettre sous la forme

Pour chaque **élément** dans la **collection**, **faire instruction**.

La syntaxe est

```

foreach(variable in collection) instruction;

```

Il est à noter que `variable` est accessible en lecture seulement.

Pour illustrer cette dernière instruction de boucle, nous allons nous pencher sur deux exemples. Le premier est celui concernant le changement de dimension d'un tableau comme annoncé en 0 et le second est une réécriture de l'exemple présenté en 5.7.

Nous avons

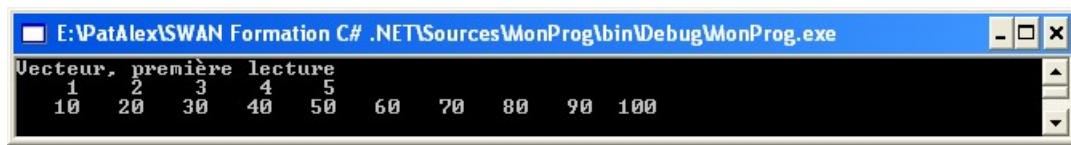
```

using System;

class MonProg
{
    static void Main()
    {
        int[] vec = {1,2,3,4,5};
        Console.WriteLine("Vecteur : Première lecture");
        foreach(int i in vec)
            Console.Write("{0,5}", i);
        Console.WriteLine();
        vec = new int[] {10,20,30,40,50,60,70,80,90,100};
        Console.WriteLine("Vecteur : Deuxième lecture");
        foreach(int i in vec)
            Console.Write("{0,5}", i);
        Console.WriteLine();
        Console.ReadLine();
    }
}

```

L'exécution fournit, heureusement,



Le second exemple peut, quant à lui, se mettre sous la forme

```
using System;

class MonProg
{
    static void Main(string[] args)
    {
        int i=1;
        foreach(string s in args)
        {
            Console.WriteLine("Argument {0} : {1}", i++, s);
            Console.ReadLine();
        }
    }
}
```

5.11. Les instructions de saut

5.11.1. Le saut classique

Les instructions de saut sont

- `break` pour terminer la boucle en cours,
- `continue` pour passer à l'itération suivante dans la boucle en cours,
- `goto` pour effectuer un saut vers une instruction *étiquetée*.²²

Pour illustrer les instructions `continue` et `break`, nous pouvons rédiger un petit bout de programme permettant de dénombrer les caractères et les lignes encodées par l'utilisateur :

```
using System;

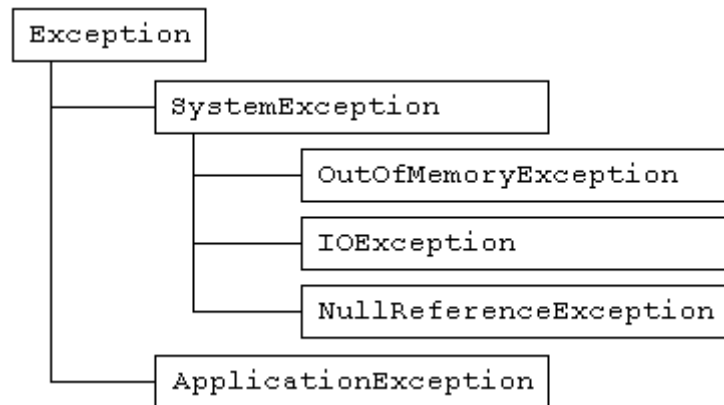
class MonProg
{
    static void Main()
    {
        int nbC=0, nbL=0;
        bool flL=true;
        char c;
        Console.WriteLine("Encodage");
        for(int k=Console.Read(); k!=-1; k=Console.Read())
        {
            c=(char)k;
            if(c==' ' || c=='\t' || c=='\r') continue;
            if(c=='\n')
            {
                if(flL) break;
                nbL++;
                flL=true;
            }
            else
            {
                nbC++;
                flL=false;
            }
        }
        Console.WriteLine("Nombre de caractères : {0}", nbC);
        Console.WriteLine("Nombre de lignes : {0}", nbL);
        Console.ReadLine();
    }
}
```

²² Nous **déconseillons fortement** l'utilisation de cette instruction qui rend très rapidement les programmes inextricables.

5.11.2. Le saut sur erreur

Une fois la logique du programme conçue, le programmeur doit se pencher sur la gestion des exceptions ou encore des erreurs qui peuvent survenir pour une raison ou pour une autre. Le respect des propriétés de robustesse et de fiabilité exigent la mise en place de cette gestion. Le problème essentiel est que cette prise en charge rend assez rapidement le code illisible. Les langages de programmation moderne facilitent cette tâche. Nous allons maintenant voir les instructions relatives aux exceptions en C#.

.NET Framework met à notre disposition un ensemble de classes d'exceptions dont la racine est `Exception`. Nous pouvons représenter la hiérarchie comme suit :



Le code pouvant subir un problème à l'exécution reste *entier* au sein d'une clause `try`. Cela permet de faciliter la maintenance du code puisque le code *normal* n'est pas entrecoupé par la gestion des erreurs et que toute la gestion de l'erreur s'en retrouve *centralisée*. C'est un apport indéniable par rapport au langage C.

Le code relatif au comportement de l'application est inséré dans des clauses suivantes, introduites par

- `catch` qui prend en charge les erreurs prévues par .NET,
- `throw` pour gérer les exceptions de manière personnalisée,
- `finally` reprend les instructions devant être de toute manière exécutées.

La syntaxe est la suivante

```
try
{
    // Logique du programme
    throw new ObjetDérivéException
}
catch (IdentificateurDeClasse variable)
{
    // Gestion des erreurs
}
finally
{
    // Code à exécuter, de toute manière
}
```

l'objet appelé par `throw` étant un objet dérivant de la classe `Exception` mise à notre disposition par C# et implémentée par nos soins.

Pour illustrer l'utilisation de ces instructions, nous allons nous baser sur du code.

Ainsi, nous pouvons écrire l'application suivante qui demande à l'utilisateur d'encoder un nombre entier et

- gère les encodages en dehors des limites acceptables pour le type `int`,
- ne traite pas les nombres entiers négatifs.

Nous pouvons écrire

```
using System;

class MonException:Exception
{public MonException(string message):base(message) {}
}

class MonProg
{static void Main()
{int i;
try
{Console.WriteLine("Un nombre : ");
i=int.Parse(Console.ReadLine());
if(i<0) throw new MonException("Que des entiers positifs !");
}
catch (OverflowException)
{Console.WriteLine("Nombre hors limite");
}
catch (MonException me)
{Console.WriteLine(me.Message);
}
finally
{Console.Read();
}
}
}
```

Signalons, pour terminer ce paragraphe sur le traitement d'erreurs que, par défaut, le dépassement de capacité arithmétique n'est pas surveillé. Par exemple, si nous demandons d'ajouter deux entiers valant 2×10^9 , le résultat sera évalué en repassant par les négatifs, phénomène connu des programmeurs C. L'exception de dépassement (`OverflowException`) ne se déclenche donc pas.

Nous pouvons exiger cette vérification par la commande `checked`. Cette dernière peut porter²³ sur un groupe d'instruction ou sur une expression arithmétique.

23 Il est également possible d'activer ce type de vérification pour l'entièreté de l'application par l'option `/checked+` lors de la compilation en ligne de commande.

Nous pouvons écrire

```
using System;

class MonProg
{
    static void Main()
    {
        checked
        {
            int nombre = int.MaxValue;
            Console.WriteLine(++nombre);
        }
    }
}
```

pour déclencher l'exception arithmétique de dépassement de capacité et éviter ainsi des comportements difficiles à corriger²⁴.

Par contre, le code suivant

```
using System;

class MonProg
{
    static void Main()
    {
        unchecked
        {
            int nombre = int.MaxValue;
            Console.WriteLine(++nombre);
        }
    }
}
```

va renvoyer la valeur -2147483648.

Ce type d'écriture peut également se faire uniquement sur l'opération arithmétique devant être vérifiée. La syntaxe est la suivante

```
using System;

class MonProg
{
    static void Main()
    {
        int nombre = int.MaxValue;
        Console.WriteLine(checked(++nombre));
    }
}
```

24 Il est clair que lorsque le produit de deux nombres positifs donne un nombre négatif, l'erreur est facilement décelable, en connaissant le phénomène de *boucle* sur les valeurs possibles déjà rencontré en C. Par contre, lorsque ce problème de signe n'apparaît pas ...

6. Les fonctions (méthodes)

Comme indiqué dans le titre, la notion de fonction est inappropriée en C# et doit être remplacée systématiquement par la notion de méthode. En effet, la notion de classe est omniprésente et une fonction ne peut exister indépendamment d'une classe.

6.1. Définition

Comme d'habitude, une méthode n'est rien d'autre que le regroupement, sous un certain vocable, de plusieurs instructions. Il convient *simplement* de déterminer

- la portée (`public`, `private`, ...),
- le type de retour (`int`, `double`, `void`, ...),
- le nom,
- les paramètres et leur type,
- les instructions,
- l'éventuelle valeur de retour (`return`).

Au niveau de la définition, nous ne noterons guère de différence entre les méthodes C# et les fonctions C. Ainsi, une méthode ne peut être définie au sein d'une autre méthode. Par contre, la notion de prototype est laissée au placard.

La différence avec C est plus philosophique et provient de l'adaptation à ce que les classes règnent en maître et que nous allons travailler avec des méthodes associées à des classes.

Nous sommes maintenant confrontés à la mise en place de l'équation présentée en 1.4 :

Méthodes + Données = Objets.

La méthode `Main()` rencontrée jusqu'à présent n'est rien d'autre qu'une méthode. Sa particularité réside dans le fait qu'il s'agit du point d'entrée de l'application, raison pour laquelle elle est affublée du qualificatif `static`.

Nous avons également utilisé d'autres méthodes dans les applications vues auparavant : `WriteLine()`, `Write()`, `ReadLine()` et `Read()`. Elles sont associées à la classe `Console` dont la définition se trouve dans l'espace de noms `System`. Nous retrouvons, dans cette notion d'espace de noms, la philosophie *étendue* des fichiers `#include` du C.

6.2. Utilisation

Par l'utilisation des méthodes de la classe `Console`, nous savons comment appeler de telles méthodes.

En ce qui concerne l'utilisation des méthodes associées aux classes, signalons que

- une méthode peut être appelée au sein de la classe où elle est définie,
- une méthode ne peut être appelée au sein d'une autre classe que dans le cas où elle est qualifiée de `public`²⁵ et il faut spécifier la classe à laquelle elle appartient.

Si besoin est, nous pouvons transmettre des arguments à la méthode. Les méthodes peuvent être surchargées. Pour rappel, cela signifie qu'une même méthode est définie pour plusieurs utilisations possibles, variant en fonction du nombre et du type de ses arguments.

Par exemple, la méthode `WriteLine()` de la classe `Console` ne compte pas moins de 19 définitions, comme on peut le voir lors de la dactylographie de l'instruction

```
Console.WriteLine(|  
▲ 1 sur 19 ▼ void Console.WriteLine (string format, params object[] arg)  
format: Chaîne de format.
```

Ces versions se distinguent uniquement par les arguments qu'il est possible de transmettre :

```
WriteLine();  
WriteLine(bool);  
WriteLine(char);  
WriteLine(char[]);  
WriteLine(decimal);  
WriteLine(double);  
WriteLine(int);  
WriteLine(long);  
WriteLine(object);  
WriteLine(float);  
WriteLine(string);  
WriteLine(uint);  
WriteLine(ulong);  
WriteLine(string, object);  
WriteLine(string, params object[]);  
WriteLine(char[], int, int);  
WriteLine(string, object, object);  
WriteLine(string, object, object, object);  
WriteLine(string, object, object, object, object);
```

²⁵ Les mots clés `protected` et `internal` permettent également d'étendre la vue d'une méthode à d'autres classes que celle où elle est définie, conformément à ce qui est présenté en 7.1.1.

6.3. Les instructions associées à la méthode

Toute variable déclarée au sein d'une méthode lui est locale. Les variables globales disparaissent et font place aux variables partagées qui ne sont rien d'autre que des variables définies au sein de la classe et accessibles par toute méthode de cette dernière.

Les conflits de portée sont résolus en donnant la priorité aux variables locales.

Ne pas oublier l'instruction `return` pour retourner une réponse à l'utilisation de la méthode. Dans ce cas, la déclaration de la méthode se fait en renseignant le type renvoyé. Par exemple,

```
using System;

class MonProg
{
    private static int Facto(int a)
    {
        int rep=1;
        if(a<0) return -1;
        else if(a<2) return 1;
        else checked
        {
            try
            {
                for(int i=2;i<=a;i++) rep*=i;
            }
            catch (OverflowException)
            {
                Console.WriteLine("Résultat hors limites !");
                rep=-1;
            }
            return rep;
        }
    }

    static void Main()
    {
        int x=8;
        Console.WriteLine("La factorielle de {0} donne {1}.",x,Facto(x));
        Console.ReadLine();
    }
}
```

Une des différences les plus marquantes par rapport au C dans le domaine des fonctions (méthodes) est que les prototypes (et les fichiers renseignés par `#include`) sont abandonnés. Cette nuance a été annoncée en 2.2.1.

6.4. Mode de transmission d'arguments

Nous pouvons maintenant aborder la gestion des paramètres.

Deux modes de transmission d'arguments sont en vigueur, à savoir le passage par valeur et par référence. Ce dernier est en fait le mode par adresse connu du C mais utilisé de manière plus souple. Il s'agit donc bien de manipuler des adresses ...

Nous pouvons ajouter les paramètres de sortie permettant de gérer une méthode qui *retourne* plusieurs valeurs.

Nous pouvons représenter schématiquement la situation de la manière suivante :

entrée	Passage par valeur Les données sont transmises à l'intérieur de la méthode mais non à l'extérieur (une modification à l'intérieur n'est pas répercutée à l'extérieur).
entrée-sortie	Passage par référence Les données sont transmises à l'intérieur de la méthode et les modifications sont répercutées en dehors.
sortie	Paramètres de sortie Les données sont uniquement transmises à l'extérieur de la méthode.

Nous pourrions également parler du mode de transmission par adresse comme nous l'avons rencontré en C. Néanmoins, l'utilisation de pointeurs, si elle est encore possible, est fortement déconseillée puisque le mode de travail devient alors *unsafe*, c'est tout dire ...

6.5. Le passage d'arguments par valeur

Nous sommes ici confrontés au mode de transmission par défaut. D'ailleurs, c'est celui que nous avons rencontré jusqu'à présent.

Les arguments sont pris en charge par les paramètres qui contiennent une copie de ce qui a été transmis. Toute modification sur les paramètres de la fonction peut se faire mais ne sera pas répercutée au niveau des arguments.

L'exemple classique permettant de distinguer le passage par valeur de celui par référence est celui de l'échange de valeurs.

Ainsi, nous pouvons écrire

```
using System;

class MonProg
{
    private static void Switch(int a, int b)
    {
        int tmp=a;
        a=b;
        b=tmp;
        Console.WriteLine(" -> Vérification : {0} et {1}.", a, b);
    }

    static void Main()
    {
        int x=8, y=12;
        Console.WriteLine("Valeurs : {0} et {1}.", x, y);
        Switch(x, y);
        Console.WriteLine("Valeurs : {0} et {1}.", x, y);
        Console.ReadLine();
    }
}
```

Le résultat à l'exécution est le suivant



```
E:\PatAlex\SWAN Formation C# .NET\Sources\MonProg\bin\Debug\MonProg.exe
Valeurs : 8 et 12.
-> Vérification : 12 et 8.
Valeurs : 8 et 12.
```

6.6. Le passage d'arguments par référence

Pour prévenir C# de l'utilisation d'arguments passés par référence, il suffit d'utiliser le mot réservé `ref`, aussi bien du côté de l'appel que de celui de la définition. Comme annoncé, dans ce cas, nous utilisons un processus semblable à celui des pointeurs de C et faisons dès lors référence à l'adresse des variables.

En reprenant ce qui précède, nous pourrions avoir

```
using System;

class MonProg
{private static void Switch(ref int a,ref int b)
{int tmp=a;
 a=b;
 b=tmp;
 Console.WriteLine(" -> Vérification : {0} et {1}.",a,b);
}

static void Main()
{int x=8,y=12;
 Console.WriteLine("Valeurs : {0} et {1}.",x,y);
 Switch(ref x,ref y);
 Console.WriteLine("Valeurs : {0} et {1}.",x,y);
 Console.ReadLine();
}
}
```

Dans ce cas de figure, le résultat à l'exécution devient



6.7. Les paramètres de sortie

Le principe qui régit le fonctionnement des arguments de sortie est identique à celui qui concerne la transmission par référence. La seule différence réside dans le fait que la transmission ne se fait pas vers la méthode.

C# met ce mécanisme en place par l'intermédiaire du mot réservé `out` comme dans l'exemple qui suit.

```
using System;

class MonProg
{private static void Initialiser(out int a)
{a=15;
}

static void Main()
{int x;
 Initialiser(out x);
 Console.WriteLine("Valeur : {0}.",x);
 Console.ReadLine();
}
}
```

Nous avons vu qu'une variable ne pouvait être utilisée tant qu'une valeur ne lui était pas assignée. Passer une variable comme argument d'une méthode est considéré comme une application de cette règle. Ce mode de transmission permet de contourner cette contrainte.

A l'aide de ce type de paramètre, nous pouvons modifier la méthode `Facto()` vue auparavant. Plutôt que de répondre `-1` lorsque la factorielle ne peut être calculée, nous allons utiliser une valeur de retour booléenne comme témoin du bon fonctionnement de la méthode, la réponse éventuelle étant stockée dans un paramètre en sortie. Ainsi, nous pouvons écrire

```
using System;

class MonProg
{
    private static bool Facto(int n, out int rep)
    {
        if (n < 0)
        {
            rep = 0;
            return false;
        }
        else if (n < 2)
        {
            rep = 1;
            return true;
        }
        else checked
        {
            try
            {
                rep = 1;
                for (int i = 2; i <= n; i++)
                    rep *= i;
                return true;
            }
            catch (OverflowException)
            {
                rep = 0;
                return false;
            }
        }
    }

    static void Main()
    {
        int x = 8, y;
        if (Facto(x, out y)) Console.WriteLine("La factorielle de {0} donne {1}.", x, y);
        Console.ReadLine();
    }
}
```

6.8. Arguments en nombre variable

Supposons que nous souhaitions créer une fonction affichant un certain nombre d'entiers transmis en arguments, ce nombre étant inconnu à l'avance.

Le mot réservé `params` permet de se sortir de cette *impasse*. Nous pouvons écrire

```
using System;

class MonProg
{
    private static void Moyenne(params int[] vec)
    {
        int som = 0, nb;
        for (nb = 0; nb < vec.Length; nb++)
            som += vec[nb];
        Console.WriteLine("Moyenne des {0} nombres : {1}", nb, (float)som/nb);
    }

    static void Main()
    {
        Moyenne(1, 2, 3, 4);
        Moyenne(2, 4, 6, 8, 10, 12, 14, 16, 18, 20);
        Console.ReadLine();
    }
}
```

La transmission des arguments à la fonction peut également se faire par l'intermédiaire d'un tableau de la manière suivante

```
• Moyenne(new int[] {1,2,3,4});  
• Moyenne(new int[] {2,4,6,8,10,12,14,16,18,20});
```

6.9. La récursivité

Rien de spécial à signaler si ce n'est que les méthodes peuvent être mutuellement récursives :

la méthode A () appelle la méthode B () qui appelle la méthode A ()

et ainsi de suite.

Pour illustrer le principe de récursivité, nous pouvons revenir à l'exemple classique de la factorielle d'un nombre développé en 6.3. Nous pouvons écrire, sous forme récursive,

```
• using System;  
•  
• class MonProg  
• {private static int Facto(int a)  
• {if (a<0) return -1;  
•   else if (a<2) return 1;  
•     else checked  
•       {try  
•         {return a*Facto(a-1);  
•       }  
•       catch (OverflowException)  
•       {Console.WriteLine("Résultat hors limites !");  
•         return -1;  
•       }  
•     }  
• }  
• static void Main()  
• {int x=8;  
•   Console.WriteLine("La factorielle de "+x+" vaut "+Facto(x));  
•   Console.ReadLine();  
• }  
• }
```

6.10. La surcharge de méthode

Nous avons vu en 6.2 que la méthode WriteLine() de la classe Console était quelque peu surchargée. Nous pouvons utiliser cette possibilité pour développer nos propres ressources.

Supposons travailler sur la structure présentée en 4.5, à savoir

```
• struct etud  
• {public string nom;  
•   public int cote;  
• }
```


Nous allons créer une méthode `Modifier()` permettant de changer les données concernant le nom, la cote ou, finalement, les deux. Nous pouvons compléter la définition de notre structure comme suit

```
public void Modifier(string n)
{
    nom=n;
    Console.WriteLine("Modifier string");
}
public void Modifier(int c)
{
    cote=c;
    Console.WriteLine("Modifier int");
}
public void Modifier(string n,int c)
{
    nom=n;
    cote=c;
    Console.WriteLine("Modifier string, int");
}
```

La distinction entre les différentes méthodes *homonymes* se fait selon les arguments reçus.

Pour vérifier cette affirmation, nous pouvons créer ma méthode `Main()` suivante, en supposant disposer des définitions présentées auparavant.

```
static void Main()
{
    etud eleve;
    eleve.nom="Winch";
    eleve.cote=17;
    Console.WriteLine("{0} a obtenu {1}/20",eleve.nom,eleve.cote);
    eleve.Modifier("Winnie");
    Console.WriteLine("{0} a obtenu {1}/20",eleve.nom,eleve.cote);
    eleve.Modifier(14);
    Console.WriteLine("{0} a obtenu {1}/20",eleve.nom,eleve.cote);
    eleve.Modifier("Tintin",20);
    Console.WriteLine("{0} a obtenu {1}/20",eleve.nom,eleve.cote);
}
```

Nous obtenons alors le résultat

```
Winch a obtenu 17/20
Modifier string
Winnie a obtenu 17/20
Modifier int
Winnie a obtenu 14/20
Modifier string, int
Tintin a obtenu 20/20
```

6.11. Les méthodes et les tableaux

Le passage d'un tableau en argument d'une fonction se fait par référence mais il n'est pas nécessaire de le signaler par le mot réservé `ref`. Ainsi, le programme suivant permet de modifier les valeurs stockées dans le tableau `vecteur` transmis en argument.

```
using System;

class MonProg
{
    static void ManipulerTableau(int[] table)
    {
        for(int i=0;i<table.Length;i++) table[i]*=2;
    }
    static void Main()
    {
        int[] vecteur = {0,5,10,15,20};
        ManipulerTableau(vecteur);
        foreach(int elt in vecteur)
            Console.WriteLine(elt);
    }
}
```

Rappelons au passage la distinction qui existe entre les boucles `foreach` et `for`. La première ne peut modifier les valeurs qu'elle manipule, contrairement à ce qui se passe avec la seconde.

Si le tableau n'est pas initialisé lors de l'appel de la fonction et que cette dernière prend en charge cette initialisation, il suffit d'utiliser le mot réservé `out` comme présenté auparavant. Ainsi, nous pouvons écrire

```
using System;

class MonProg
{
    static void RemplirTableau(out int[] table)
    {
        for(int i=0;i<5;i++) table[i] = 5*(1+i);
    }
    static void Main()
    {
        int[] vecteur;
        RemplirTableau(out vecteur);
        foreach(int elt in vecteur)
            Console.WriteLine(elt);
    }
}
```

Pour terminer la collaboration entre les tableaux et les fonctions, nous allons maintenant examiner comment il est possible, pour une fonction, de renvoyer un tableau en réponse. Pour ce faire, examinons le code source suivant

```
using System;

class MonProg
{
    static int[] CreerTableau()
    {
        int[] temporaire = {0,5,10,15,20};
        return temporaire;
    }
    static void Main()
    {
        int[] vecteur;
        vecteur = CreerTableau();
        foreach(int elt in vecteur)
            System.Console.WriteLine(elt);
    }
}
```

La référence temporaire est locale à la fonction `CreerTableau()` mais le déroulement des opérations est quelque peu particulier. Le tableau temporaire est initialisé et renseigne l'adresse du premier élément du tableau créé (une notion de pointeur...), adresse renvoyée comme réponse de la fonction. La référence temporaire étant considérée comme locale à la fonction est appelée à disparaître lors de la sortie de la fonction, et seulement la référence et non les 5 éléments créés.

6.12. Les délégués

Un délégué est un objet qui permet d'appeler une fonction. Une fois déclaré, l'objet pourra faire référence à des méthodes différentes possédant la signature²⁶ qui lui est assignée. Un tel objet est semblable à un pointeur de fonctions disponibles dans le langage C.

26 Par signature, nous entendons valeur éventuelle de retour et paramètres transmis.

Dans l'exemple suivant, nous allons créer deux méthodes `Double` et `Triple` qui ne renvoient pas de valeur mais attendent un entier en paramètre. Ensuite, nous définissons un objet `Traiter` de type `delegate`. Il s'agit donc d'un délégué pour des méthodes qui ne renvoient rien (`void`) et ont un paramètre de type entier (`int`). Une fois les ressources mises en place, nous pouvons, dans la méthode `Main()`, déclarer une variable `t` de type `Traiter` et l'instancier successivement aux fonctions `Double` et `Triple`. Un appel à cet objet `t` entraîne en fait l'appel à la méthode *pointée*.

```
using System;

class Program
{
    static private void Double(int a)
    { Console.WriteLine("Double de " + a.ToString() + " = " + (2 * a).ToString()); }
    static private void Triple(int a)
    { Console.WriteLine("Triple de " + a.ToString() + " = " + (3 * a).ToString()); }
    delegate void Traiter(int b);
    static void Main()
    {
        Traiter t = new Traiter(Double);
        t(5);
        t = new Traiter(Triple);
        t(5);
    }
}
```

A l'exécution, le résultat est semblable à

```
Le double de 5 vaut 10
Le triple de 5 vaut 15
-
```

Signalons également qu'il est possible de *chaîner* les méthodes *pointées* par un délégué. Les opérateurs `+=` et `-=` permettent, respectivement, d'ajouter et de supprimer une méthode à la *liste*. La particularité d'une telle *construction* est que l'exécution du délégué provoquera l'exécution de toutes les méthodes ainsi associées.

En nous basant sur les mêmes ressources que précédemment, nous pouvons écrire la méthode `Main()` suivante.

```
static void Main()
{
    Traiter t = new Traiter(Double);
    Console.WriteLine("La seule méthode Double est exécutée");
    t(5);
    t += new Traiter(Triple);
    Console.WriteLine("La méthode Triple a été ajoutée");
    t(5);
    Console.WriteLine("La méthode Double a été supprimée");
    t -= new Traiter(Double);
    t(5);
}
```

Le résultat affiché à l'écran sera le suivant

```
La seule méthode Double est exécutée
Le double de 5 vaut 10
La méthode Triple a été ajoutée
Le double de 5 vaut 10
Le triple de 5 vaut 15
La méthode Double a été supprimée
Le triple de 5 vaut 15
-
```

6.13. Les méthodes anonymes

Le langage C# permet, depuis sa version 2, de définir des instructions sous la forme de méthodes sans être obligé de créer explicitement la méthode. Cette manière de faire est utilisée via les délégués qui ne font donc plus forcément référence à une méthode existante.

Nous pouvons reprendre le premier exemple du paragraphe précédent et le compléter par une méthode qui calcule le quadruple du nombre transmis en paramètre, cette méthode n'existant pas explicitement si ce n'est sous la forme du code associé. Nous avons

```
using System;

class Program
{
    static private void Double(int a)
    { Console.WriteLine("Double de " + a.ToString() + " = " + (2 * a).ToString()); }
    static private void Triple(int a)
    { Console.WriteLine("Triple de " + a.ToString() + " = " + (3 * a).ToString()); }
    delegate void Traiter(int b);
    static void Main()
    {
        Traiter t = new Traiter(Double);
        t(5);
        t = new Traiter(Triple);
        t(5);
        t = delegate(int b)
        { Console.WriteLine("Quadruple de " + b.ToString() + " = " + (4 * b).ToString()); };
        t(5);
    }
}
```

Le résultat sera le suivant

```
Double de 5 = 10
Triple de 5 = 15
Quadruple de 5 = 20
```

Evidemment, comme les méthodes *pointées* ne sont pas *nommées* dans le cas de méthodes anonymes, même s'il reste possible de les ajouter à la liste des méthodes pointées par un objet de type `delegate`, il n'est pas possible de les en retirer.

6.14. Les expressions lambda

La version 3 permet de simplifier l'utilisation des méthodes anonymes via les expressions lambda.

La définition d'une expression lambda se fait en deux étapes. La première partie consiste en la liste des paramètres et la seconde, séparée de la première par le symbole `=>`, contient une expression destinée à être évaluée, une instruction devant être exécutée.

Nous complétons le code précédent en nous appuyant sur le délégué `t` défini en 6.13. Le code suivant permet de définir une expression lambda qui permet d'afficher le quintuple de l'argument transmis lors de l'appel.

```
t = b => { Console.WriteLine("Quintuple de " + b.ToString() + " = " + (5 * b).ToString()); }
```

En demandant l'exécution de `t(5)`, nous obtenons le résultat suivant à l'écran.

```
Quintuple de 5 = 25
```

6.15. Les méthodes génériques

Une méthode générique est une méthode qui est déclarée avec des paramètres de type. Cela permet à la méthode de s'adapter aux arguments transmis en ne limitant pas à un seul type ces arguments.

Par exemple, nous pouvons écrire

```
static void Permuter<T>(ref T argument1, ref T argument2)
{
    T tmp = argument1;
    argument1 = argument2;
    argument2 = tmp;
}
```

Cette méthode vise à permuter les valeurs des arguments. Pour que cela ait une répercussion sur le programme appelant, il faut que nous transmettions ces arguments par adresse, ce qui explique la présence de `ref`.

Nous annonçons les types génériques utilisés par la méthode par l'intermédiaire de `<>`. Dans cet exemple, le type générique sera donc représenté par `T`.

L'appel à une telle fonction se fait de manière classique si ce n'est que, entre `<` et `>`, nous renseignons le(s) type(s) effectivement transmis. Nous pouvons écrire

```
int a = 5, b = 10;
Permuter<int>(ref a, ref b);
```

ou

```
float a = 5, b = 10;
Permuter<float>(ref a, ref b);
```

Notons toutefois que le renseignement de type n'est pas obligatoire. Il sera automatiquement déduit selon les arguments transmis. Evidemment, ce *raccourci* est à oublier lorsqu'aucun argument n'est transmis à la méthode.

7. Les classes

7.1. Introduction

Il s'agit de l'élément essentiel du langage C#. Nous avons pu nous en rendre compte rien qu'en développant une application *console* puisque tout notre code était inséré au sein d'une classe, *MonProg* en l'occurrence.

7.1.1. Définition des classes

Commençons par mettre en place ou rappeler les bases concernant les classes.

Les classes regroupent

- des variables qui sont appelées champs ou attributs de la classe,
- des fonctions qui sont appelées méthodes de la classe et qui sont destinées à traiter les champs de la classe,
- des propriétés et des indexeurs.

Comme pour les structures, C# nous laisse la possibilité de définir nos propres classes. Dans un premier temps, nous allons nous consacrer à définir des données, les méthodes venant par la suite. Par exemple, en nous basant sur ce que nous avons vu auparavant, nous pouvons écrire

```
class etud
{
    public string nom;
    public int cote;
}
```

Comme nous l'avons vu en 6.1 pour les méthodes, il est possible de préciser la portée des données membres des classes. Nous disposons des possibilités suivantes :

- `public` permet d'accéder au champ en toutes circonstances,
- `private` protège le champ des accès en provenance de l'extérieur de la classe de définition,
- `protected` restreint l'accès au champ aux seules classes qui héritent de la classe de définition,
- `internal` limite l'accès au champ au sein de l'espace de noms contenant la classe de définition,
- `protected internal` renseigne un accès `protected` ou `internal`.

Par rapport aux structures, nous pouvons donc ajouter les niveaux de protection `protected` et `internal protected`.

Nous rappelons un des principes de la programmation orientée objet : éviter, autant que faire se peut, de déclarer les données en tant que `public`.

7.1.2. Les objets : création

Les objets sont la réalisation du concept abstrait qu'est la classe, tout comme l'est la variable par rapport aux types primitifs. Ils sont générés en mémoire par l'intermédiaire de la commande `new`. Par exemple, nous pouvons écrire, en fonction de la déclaration de classe vue auparavant,

```
etud eleveA, eleveB;  
eleveA = new etud();  
eleveB = new etud();
```

Nous noterons l'utilisation des parenthèses, obligatoires, pour *construire* un objet, même si le constructeur utilisé ne nécessite pas d'argument. La méthode `etud()` est en fait le constructeur par défaut et est fourni d'office par C#. Nous y revenons en 7.2.2.

Un objet est une référence à un emplacement mémoire correspondant à la taille de la classe. Pour rappel, il s'agit de pointeurs déguisés. Chaque objet créé à partir de la classe `etud` possède les mêmes caractéristiques conceptuelles (attributs, propriétés) tout en étant qu'ils ont également leur propre identité, comme une variable classique.

Par conséquent, l'écriture

```
etud eleveA = new etud();  
etud eleveB = eleveA;
```

engendre la création d'un seul objet de type `etud`, à savoir `eleveA` car `eleveB` ne fait rien d'autre que de référencer le même objet. Pour réaliser de réelles copies, quelques petites modifications sont à envisager. Nous y reviendrons en 7.5.1.

De la même manière, l'écriture

```
etud eleveA = new etud(), eleveB = new etud();  
if (eleveB == eleveA) Console.WriteLine("Etudiants identiques");  
else Console.WriteLine("Etudiants différents");
```

risque de ne pas effectuer ce que nous attendons d'elle. En effet, s'appuyant sur la définition par référence, la condition `eleveB == eleveA` compare les emplacements en mémoire des deux objets et non le contenu. Nous revenons sur la comparaison des contenus en 7.5.2.

Notons aussi qu'il est possible de définir une valeur par défaut pour chaque donnée de la classe. Par exemple, nous pourrions avoir la déclaration suivante

```
class etud  
{  
    public string nom="Etudiant anonyme";  
    public int cote=0;  
}
```

Signalons enfin un cas particulier de l'initialisation des données membres d'une classe par l'intermédiaire des mots réservés `const` et `readonly`. Il s'agit, dans les deux cas, de déterminer une valeur invariable pour une donnée membre. La seule différence concerne l'endroit de l'assignation. Ainsi, le mot `const` implique que la valeur invariable soit assignée lors de la définition de la classe (à la compilation) tandis que `readonly` est applicable dans le constructeur de la classe (à l'exécution). Ces deux caractéristiques d'invariance trouvent leur origine dans le fait que les objets sont manipulés par référence, ce qui implique qu'ils sont à tout moment modifiables.

7.1.3. Les objets : la destruction

Les objets sont gérés par référence. C'est un système de pointeurs qui est sous-jacent.

En C, le programmeur doit prendre soin de remettre le système dans l'état dans lequel il l'a trouvé en libérant les ressources utilisées dynamiquement.

En C#, le système de destruction des objets est totalement pris en charge par .NET et son *garbage collector*. Lorsqu'un objet cesse d'exister, soit parce que la fonction où il a été déclaré est terminée, soit parce que la valeur `null` lui a été assignée, il est renseigné comme étant non accessible et l'emplacement mémoire qu'il occupait sera libéré dès que .NET en trouvera l'utilité.

7.1.4. Les objets : accéder aux données

Pour accéder aux données membres d'un objet, le symbole `.` est d'actualité comme c'est le cas pour les structures. Ainsi, nous pouvons écrire, toujours en nous basant sur la définition de la classe `etud` qui précède,

```
etud eleve = new etud();  
eleve.nom = "Winch";  
eleve.cote = 17;  
Console.WriteLine("Eleve "+eleve.nom+" a eu "+eleve.cote+"/20");
```

7.1.5. Imbriquer les classes

Le langage C# permet de définir des classes au sein d'une classe. Nous ne sommes d'ailleurs pas limités aux classes car, au sein d'une classe, nous pouvons inclure les définitions de type

- `class`,
- `struct`,
- `interface`,
- `enum`,
- `delegate`.

Il reste à traiter l'accès de ce type de ressource défini au sein d'une classe. Nous pouvons, par exemple, considérer la définition

```
class classe  
{  
    public class in1  
    {  
        public int nb;  
    }  
    class in2  
    {  
        public int nb;  
    }  
    public string ch;  
}
```

Dès lors, au sein du programme principal, la déclaration d'un objet de type `in1` se fait selon l'instruction

```
classe.in1 elt;
```

Nous constatons que le chemin d'accès permettant d'aboutir à la définition de la classe `in1` est requis.

L'instanciation, pour sa part, suit la règle *habituelle*

```
• classe.inl elt = new classe.inl();
```

En ce qui concerne la protection des données, nous pouvons constater que tout se passe bien avec `elt` car il dérive de `classe` qui est de type `public`. Par contre, si nous essayons d'accéder à la classe `in2` en dehors de `classe`, nous nous heurtons à une ressource de type `private` (protection par défaut) qui est, par conséquent, inaccessible en dehors de sa classe de définition.

7.2. Les méthodes d'une classe

Une méthode est une fonction associée à une classe de telle manière que les classes, de par leur association *données + méthodes*, forment un tout autonome.

Tout comme les données, elles peuvent être renseignées comme étant `public`, `protected` ou `private`. Les définitions des méthodes sont incluses dans la définition de la classe dont elles dépendent. Il n'est donc plus question des fichiers d'extension `h` ni de fonctions prototypes comme en C.

Par exemple, nous avons, en définissant toutes les ressources de la classe comme étant publiques pour nous faciliter la tâche,

```
• using System;
• class etud
• {public string nom;
•   public int cote;
•   public void Afficher()
•   {Console.WriteLine("Eleve "+nom+" a eu "+cote+"/20");
•   }
• }
• class MonProg
• {static void Main()
• {etud eleve = new etud();
•   eleve.nom = "Winch";eleve.cote = 17;
•   eleve.Afficher();
• }
• }
```

Signalons qu'il est toujours possible de définir plusieurs méthodes de même nom dans une même classe à condition qu'elles diffèrent par le nombre ou le type des arguments transmis. Nous retrouvons ainsi la propriété de polymorphisme ou, plus spécifiquement, de surcharge présentée en 1.5.3.

Nous pouvons étendre la notion de classe au programme principal telle que définie en 3.2. En effet, nous avons vu que la fonction `Main()`, point d'entrée de l'application, était déclarée dans une classe. Par conséquent, cette fonction particulière est également une méthode de la classe ainsi mise en place comme le langage C définissait une fonction comme point d'entrée du programme.

Pour changer un peu les plaisirs, nous allons utiliser cette structure de programme pour créer une méthode de la classe `MonProg` et l'appeler depuis la fonction (méthode) `Main()` en instanciant un objet de ce type.

Dans cette optique, le code devient

```
using System;

class MonProg
{
    void Afficher()
    {
        Console.WriteLine("Bonjour le monde !");
        Console.Read();
    }

    static void Main()
    {
        new MonProg().Afficher();
    }
}
```

Signalons que nous pouvons, d'une manière peut-être plus efficace, placer les instructions d'affichage dans un constructeur de la classe `MonProg` :

```
using System;

class MonProg
{
    public MonProg()
    {
        Console.WriteLine("Bonjour le monde !");
        Console.Read();
    }

    static void Main()
    {
        new MonProg();
    }
}
```

La méthode `MonProg()` porte le même nom que la classe et est une méthode particulière baptisée *constructeur*. Nous reviendrons plus en détail sur ces méthodes importantes en 7.2.2.

7.2.1. Les champs et méthodes statiques

Le principe des membres statiques d'une classe est de les rendre accessibles indépendamment de l'existence d'un quelconque objet de la classe. Les données membres existent de toute manière tandis que les méthodes sont utilisables directement. Elles sont accompagnées du qualificatif `static`.

Des règles de fonctionnement sont cependant à respecter et, finalement, elles sont assez naturelles vu la description que nous venons de faire :

- une méthode statique n'a accès qu'aux données membres statiques de la classe,
- une méthode statique peut appeler une autre méthode statique mais pas une non statique,
- une méthode non statique peut accéder aux données membres statiques et non statiques.

La classe `Math` qui, comme en JavaScript, regroupe les ressources mathématiques est en fait une classe composée de nombreuses fonctions qualifiées de `static`. L'avantage est de pouvoir accéder aux fonctions mathématiques sans devoir en manipuler les objets.

En dehors de cette utilisation dans le contexte de classe omniprésente, nous pouvons également utiliser des données de type `static` lorsque nous souhaitons que les objets dérivant d'une classe partagent tous le même renseignement.

L'exemple classique est de dénombrer les objets de la classe ayant été créés. Ainsi, nous pouvons écrire le code source suivant

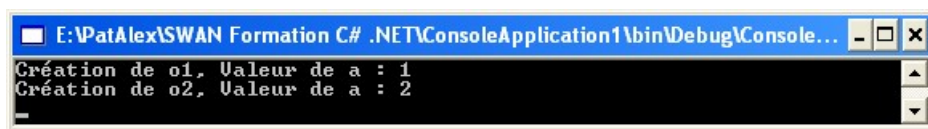
```
using System;

class MonProg
{
    class Essai
    {
        public static int a = 0;
        public Essai()
        {
            a++;
        }
    }

    static void Main()
    {
        Essai o1 = new Essai();
        Console.WriteLine("Création de o1, a vaut "+Essai.a);
        Essai o2 = new Essai();
        Console.WriteLine("Création de o2, a vaut "+Essai.a);
        Console.ReadLine();
    }
}
```

Nous noterons au passage que l'utilisation du membre statique `a` se fait par l'intermédiaire de la classe `Essai` et non d'un objet (`o1` ou `o2`) en particulier. Ceci renforce l'idée d'indépendance qui existe entre un membre statique et les différents objets qui découlent de la classe.

Le résultat à l'exécution est le suivant.



7.2.2. Les constructeurs et destructeurs de classe

Rappelons qu'il existe deux types particuliers de méthodes associés aux classes : les constructeurs et destructeurs.

Comme nous l'avons déjà signalé, les destructeurs n'ont plus à être pris en charge par le programmeur puisque .NET s'occupe complètement de la *disparition* des objets une fois ceux-ci devenus *obsoletes*. Néanmoins, nous pouvons toujours en définir.

Nous avons déjà croisé des constructeurs dans les exemples précédents. En fait, de telles méthodes sont caractérisées par

- le nom de la méthode est identique au nom de la classe,
- aucune valeur n'est renseignée en retour, pas même `void`.

En l'absence de tout constructeur, C# utilise un constructeur par défaut, sans argument. Cependant, lorsque nous mettons en place un constructeur comportant un nombre positif d'arguments, ce constructeur par défaut disparaît et l'utiliser sera signalé comme fautif. Pour pallier à cet inconvénient, nous sommes alors obligés de définir un argument par défaut, caractérisé par le fait qu'il n'a pas d'argument.

En reprenant notre exemple propre aux étudiants, nous pouvons compléter la définition de la classe `etud` en insérant un constructeur

```
class etud
{
    public string nom;
    public int cote;
    public etud(string n, int c)
    {
        nom = n;
        cote = c;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve "+nom+" a eu "+cote+"/20");
    }
}
```

Même si c'est à déconseiller, vu l'aide à la rédaction dont nous disposons²⁷, nous pouvons créer un second constructeur qui permettra à l'utilisateur de la classe de ne pas s'attarder sur l'ordre des arguments. Cependant, il faut veiller à ne pas dupliquer le code à l'infini.

La langage C# nous permet de réutiliser les méthodes déjà définies. Dans l'exemple qui suit, nous allons en profiter pour définir notre propre constructeur par défaut en nous basant sur celui défini ci-dessus.

Ainsi, plutôt que

```
class etud
{
    // Données membres et méthodes
    public etud(string n, int c)
    {
        nom = n;
        cote = c;
    }
    public etud(int c, string n)
    {
        nom = n;
        cote = c;
    }
    public etud()
    {
        nom = "Inconnu";
        cote = 0;
    }
}
```

nous préférons écrire

```
class etud
{
    // Données membres et méthodes
    public etud(string n, int c)
    {
        nom = n;
        cote = c;
    }
    public etud(int c, string n) : this(n, c)
    {
    }
    public etud() : this("Inconnu", 0)
    {
    }
}
```

²⁷ Nous avons eu l'occasion de voir que la dactylographie des méthodes nous proposait, en cours de frappe, plusieurs choix possibles lorsque la méthode est surchargée. Nous pouvons ainsi voir quels sont les arguments possibles. Evidemment, si les arguments sont de même type, cela devient plus délicat de savoir ce qu'ils représentent.

Le constructeur peut être qualifié de `static` auquel cas il est automatiquement appelé avant l'utilisation d'un champ ou d'une méthode statique de la classe. Il permet également d'initialiser les champs statiques de la classe.

Même si .NET prend en charge la suppression des objets, cela ne concerne que la gestion de la mémoire. Il est donc normal de prévoir un processus activé à la destruction d'un objet. Le développeur peut ainsi, par exemple, fermer un fichier de données associé à l'objet au moment où l'objet cesse d'exister logiquement, habituellement en sortie de bloc d'instructions. Nous évitons ainsi les éventuels problèmes liés aux accès concurrents. La syntaxe d'un destructeur est la suivante

- le nom de la méthode est identique au nom de la classe et précédé du symbole `~`,
- aucune valeur n'est renseignée en retour, pas même `void`.

Nous pouvons définir la classe

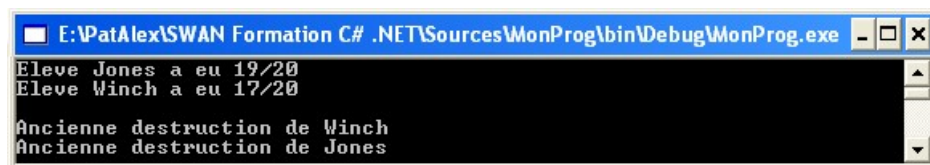
```
using System;

class etud
{
    public string nom;
    public int cote;
    public etud(string n, int c)
    {
        nom = n; cote = c;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve "+nom+" a eu "+cote+"/20");
    }
    ~etud()
    {
        Console.WriteLine("Ancienne destruction de "+nom);
    }
}
```

Le programme principal peut alors se mettre sous la forme

```
class Prog
{
    static void Main()
    {
        if(true)
        {
            etud ela = new etud("Jones", 19);
            etud elb = new etud("Winch", 17);
            ela.Afficher();
            elb.Afficher();
        }
        Console.ReadLine();
    }
}
```

Cependant, l'élimination des objets devenus obsolètes est prise en charge par .NET et le moment où s'effectue l'appel au destructeur n'est pas du ressort du développeur. Nous pouvons nous en rendre compte lors de l'exécution de ce programme qui affiche l'écran



Nous pouvons constater que, malgré le fait que la durée de vie des variables et des objets soit limitée au bloc où ils sont déclarés (instruction `if`), l'instruction de lecture au clavier est effectuée avant le nettoyage effectif de la mémoire.

Pour retrouver le mécanisme de nettoyage tel que nous le connaissons, l'environnement .NET met à notre disposition une méthode standard `Dispose()`. Pour ce faire, la classe doit hériter de la classe `IDisposable` définie dans `System`. La durée de vie d'un objet `obj` est introduite par l'expression `using (obj)` et délimitée par les traditionnelles accolades.

Le code précédent se transforme alors comme suit

```
using System;

class etud:IDisposable
{
    public string nom;
    public int cote;
    public etud(string n,int c)
    {
        nom = n;cote = c;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve "+nom+" a eu "+cote+"/20");
    }
    ~etud()
    {
        Console.WriteLine("Ancienne destruction de "+nom);
    }
    #region Membres de IDisposable
    public void Dispose()
    {
        Console.WriteLine("Destruction de "+nom);
    }
    #endregion
}
```

Le programme `Main()` devient

```
class Prog
{
    static void Main()
    {
        if(true)
        {
            etud ela = new etud("Jones",19);
            etud elb = new etud("Winch",17);
            using(ela) using(elb)
            {
                ela.Afficher();
                elb.Afficher();
            }
        }
        Console.ReadLine();
    }
}
```

L'exécution de ce programme donne



```
E:\PatAlex\SWAN Formation C# .NET\Sources\MonProg\bin\Debug\MonProg.exe
Eleve Jones a eu 19/20
Eleve Winch a eu 17/20
Destruction de Winch
Destruction de Jones
Ancienne destruction de Winch
Ancienne destruction de Jones
```

et, cette fois, le programme passe par la fonction `Dispose()` associée aux objets `elb` et `ela` avant de passer à la lecture au clavier.

7.2.3. Faire référence aux objets de la classe

Pour déterminer sans équivoque l'utilisation d'une donnée membre d'une classe, il est possible de préfixer ce dernier par le mot `this`.

Par exemple, si nous souhaitons être rigoureux, nous pouvons réécrire le constructeur de la classe `etud` de la manière suivante, quelque peu plus explicite au niveau des arguments

```
public etud(string nom,int cote)
{
    this.nom = nom;
    this.cote = cote;
}
```

Comme nous pouvons le constater, cette manière de faire permet d'utiliser le même nom qu'une donnée membre comme argument de la méthode.

7.2.4. Les propriétés

Une propriété est une méthode particulière qui s'utilise comme un champ d'une classe.

Ces fonctions, aussi appelées *accesseurs*, permettent de récupérer (*getter*) ou de modifier (*setter*) la valeur stockée dans la propriété.

Cela permet de protéger les données tout en laissant à l'utilisateur l'impression d'y accéder. En fait, comme nous allons le voir, les accesseurs (méthodes `Cote`) cachent les données de la classe (champ `cote`).

En nous basant sur la classe `etud`, nous pouvons écrire

```
class etud
{
    public string nom;
    private int cote;
    public etud(string nom,int cote)
    {
        this.nom = nom;this.cote = cote;
    }
    public int Cote
    {
        get
        {
            return this.cote;
        }
        set
        {
            if(value<0) this.cote=0;
            else if(value>20) this.cote=20;
            else this.cote=value;
        }
    }
}
```

en remarquant, au passage, que la donnée `cote` est (enfin) de type `private`. Il est évidemment possible de baptiser cette propriété différemment mais cela nuirait à la clarté du code.

L'association de champs mise en forme avec leurs accesseurs est parfois obligatoire comme nous le verrons plus tard. Cependant, il arrive qu'il n'y ait pas de traitement spécifique à mettre en place et, dans le but de réduire le code, depuis la version 3 de C#, il est possible de définir simultanément la donnée et sa propriété.

Ainsi, l'écriture suivante est permise et crée automatiquement, en *arrière-plan*, une donnée associée qui sera évidemment de même type.

```
public string Nom { get ; set ; }
```

et est équivalente à cette dernière, plus *classique*.

```
private string nom
public string Nom
{
    get { return this.nom; }
    set { this.nom=value; }
}
```

Cette manière abrégée de faire est toutefois sujette à quelques restrictions :

- les deux accesseurs (lecture-écriture) doivent être renseignés,
- les deux accesseurs ont le même niveau de protection,

7.3. L'héritage

Nous avons vu en 1.5.1 ce qu'était l'héritage et, en particulier, l'héritage multiple. En C#, comme dans d'autres langages de programmation orienté objet, nous ne disposons pas de l'héritage multiple.

Pour renseigner qu'une classe hérite des caractéristiques d'une autre classe, il suffit de faire suivre son nom directement par le symbole : suivi de la classe *mère*. Ainsi, nous pouvons nous inspirer de la classe de base `etud` définie auparavant pour déboucher sur la gestion des étudiants de 3^{ème} année en considérant l'option choisie et l'entreprise où se déroule le stage comme données supplémentaires. Nous obtenons

```
class etud3TI : etud
{
    public string option, stage;
    public etud3TI(string n, int c, string o, string s) : base(n, c)
    {
        this.option = o;
        this.stage = s;
    }
}
```

L'appel du constructeur de la classe (`etud`) dont `etud3TI` dérive se fait par l'intermédiaire de l'instruction du mot clé `base`. Ce dernier peut être absent si un constructeur sans argument est présent dans la classe de base. Sinon, un message d'erreur sera généré car l'appel du constructeur de la classe `etud` est automatique et nous retompons alors sur les règles classiques présentées en 7.2.2.

Si nous ne souhaitons créer une classe ne pouvant être héritée, il suffit d'introduire le mot réservé `sealed` dans sa définition, comme suit

```
sealed class etud3TI : etud
{
    // Définition des membres et méthodes
}
```

Par défaut, les structures sont qualifiées de `sealed`. Cette définition peut apparaître comme étant naturelle puisque les structures ne supportent pas l'héritage.

7.3.1. La surcharge de méthode

Pour illustrer ces différences, nous repartons de la classe `etud3TI` définie en auparavant. Nous allons personnaliser l'affichage des renseignements d'un objet quelconque en écrivant

```
class etud3TI : etud
{
    public string option, stage;
    public etud3TI(string n, int c, string o, string s) : base(n, c)
    {
        this.option = o;
        this.stage = s;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve " + this.nom + " a eu " + this.cote + "/20");
        Console.WriteLine("    suit l'option " + this.option);
        Console.WriteLine("    effectue son stage chez " + this.stage);
    }
}
```


Ici, le compilateur réagit avec un message d'avertissement car la fonction `afficher()` de `etud3TI` cache la fonction de la classe `etud`, avec le même nombre d'arguments.

Pour éviter ce *désagrément*, il suffit de qualifier la méthode `afficher()` de `new` :

```
class etud3TI : etud
{
    public string option, stage;
    public etud3TI(string n, int c, string o, string s) : base(n, c)
    {
        this.option = o;
        this.stage = s;
    }
    public new void Afficher()
    {
        Console.WriteLine("Eleve " + this.nom + " a eu " + this.cote + "/20");
        Console.WriteLine("    suit l'option " + this.option);
        Console.WriteLine("    effectue son stage chez " + this.stage);
    }
}
```

Histoire de raccourcir notre code, nous pouvons faire référence, dans la méthode `Afficher()` de la classe `etud3TI`, à la méthode `Afficher()` de la classe `etud`. Il suffit, pour ce faire, d'utiliser le mot réservé `base`. Ainsi, nous pouvons écrire

```
public new void Afficher()
{
    base.Afficher();
    Console.WriteLine("    suit l'option " + this.option);
    Console.WriteLine("    effectue son stage chez " + this.stage);
}
```

Notons que cet appel ne peut se faire que par rapport à la classe de *niveau* directement supérieur car l'écriture `base.base.Methode()` est rejetée à la compilation.

L'environnement des méthodes statiques est défini lors de l'exécution du programme. Par conséquent, si nous déclarons une classe qui inclut une méthode statique, puis en dérivons une nouvelle classe, la classe dérivée partage exactement la même méthode située à la même adresse. Dès lors, il est impossible de redéfinir les méthodes statiques.

7.3.2. Les méthodes virtuelles

Une méthode virtuelle est une méthode déclarée à l'aide du mot réservé `virtual`. Il est alors possible, mais non obligatoire, de redéfinir ce type de méthode dans les classes dérivées.

Les classes virtuelles proviennent d'un cas de figure particulier qui se produit lors de l'utilisation du polymorphisme. Nous allons l'illustrer par l'exemple suivant.

Nous commençons par nous baser sur les définitions

```
class etud
{
    public string nom;
    public int cote;
    public etud(string nom, int cote)
    {
        this.nom = nom; this.cote = cote;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve " + nom + " a eu " + cote + "/20");
    }
}
```

```

class etud3TI : etud
{
    public string option, stage;
    public etud3TI(string n, int c, string o, string s) : base(n, c)
    {
        this.option = o;
        this.stage = s;
    }
    public void Afficher()
    {
        Console.WriteLine("Eleve "+this.nom+" a eu "+this.cote+"/20");
        Console.WriteLine("    suit l'option "+this.option);
        Console.WriteLine("    effectue son stage chez "+this.stage);
    }
}

```

Nous pouvons alors écrire

```

class Prog
{
    static void Main()
    {
        etud ela, elb;
        etud3TI el3TI;
        Ela = new etud("Asterix", 16);
        El3TI = new etud3TI("Jones", 19, "CliSer", "ULg");
        Elb = new etud3TI("Winch", 17, "Réseau", "Magottaux");
        Ela.Afficher();
        El3TI.Afficher();
        Elb.Afficher();
    }
}

```

Deux objets ne vont poser aucun problème, à savoir `ela` et `el3TI` puisque, déclarés en tant que `etud` et `etud3TI`, ils sont utilisés en tant que tels.

Par contre, l'élément `elb` est nettement plus perturbateur. En effet, ce dernier est déclaré en tant que `etud` et va devenir `etud3TI` à sa création effective. Considérant qu'un étudiant de dernière année est un étudiant, C# accepte cette manipulation. Néanmoins, lorsque le compilateur rencontre l'instruction

```

elb.Afficher();

```

il associe la classe `etud` à `elb`, ignorant les manipulations qui ont été effectuées à la création en s'en tenant à la déclaration même de l'objet. Par conséquent, la méthode `Afficher()` qui est appelée par `elb` est celle de la classe `etud` et non celle de la classe `etud3TI` et, par conséquent, l'option et l'entreprise hôte du stagiaire ne seront pas affichées.

Pour pallier à cet inconvénient, les méthodes virtuelles interviennent. En fait, l'utilisation de telles ressources se fait en deux étapes :

- la méthode de la classe de base est qualifiée par `virtual`,
- la méthode de la classe dérivée est qualifiée par `override`.

Nous apportons donc les corrections suivantes

```

class etud
{
    // Données membres et méthodes
    public virtual void Afficher()
    {
        Console.WriteLine("Eleve "+nom+" a eu "+cote+"/20");
    }
}

```

et

```

class etud3TI : etud
{
    // Données membres et méthodes
    public override void Afficher()
    {
        Console.WriteLine("Eleve "+this.nom+" a eu "+this.cote+"/20");
        Console.WriteLine("    suit l'option "+this.option);
        Console.WriteLine("    effectue son stage chez "+this.stage);
    }
}

```

7.3.3. Les classes abstraites

Une classe abstraite contient les définitions des méthodes mais pas leur implémentation. Les méthodes définies dans une telle classe devront être implémentées dans les classes filles en utilisant le mot clé `override`. En fait, en C#, une méthode abstraite est automatiquement virtuelle.

Cela correspond à la philosophie de la programmation orientée objet. En effet, si nous considérons les classes `rac`, `dera` et `derb` où `dera` et `derb` dérivent de `rac`, la modélisation peut souhaiter que la classe `rac` contienne les définitions communes à toutes les classes qui en dérivent, à savoir `dera` et `derb`. La plupart du temps, seule la définition est importante au niveau de `rac` puisque, étant plus riches, les classes dérivées `dera` et `derb` vont devoir la personnaliser pour l'adapter à leur propre spécificité.

Les règles concernant de telles classes sont

- la classe doit être qualifiée par le mot réservé `abstract`,
- la classe ne peut être instanciée ou, de manière équivalente, aucun objet ne peut être créé à partir d'une classe abstraite.

Nous pouvons reprendre l'exemple précédent et écrire

```
abstract class etud
{
    public string nom;
    public int cote;
    public etud(string nom, int cote)
    {
        this.nom=nom; this.cote=cote;
    }
    public abstract void Afficher();
}

class etud3TI : etud
{
    public string option, stage;
    public etud3TI(string n, int c, string o, string s) : base(n, c)
    {
        this.option=o; this.stage=s;
    }
    public override void Afficher()
    {
        Console.WriteLine("Eleve "+this.nom+" a eu "+this.cote+"/20");
        Console.WriteLine("    suit l'option "+this.option);
        Console.WriteLine("    effectue son stage chez "+this.stage);
    }
}
```

7.3.4. Les interfaces

Les interfaces ne possèdent aucune donnée membre et aucune implémentation de méthode. Tout ce qu'elles contiennent sont les tâches devant être effectuées par les classes dérivées.

Comme c'est le cas pour les classes abstraites, une interface ne peut être instanciée et les classes qui dérivent une interface doivent implémenter toutes les méthodes reprises dans l'interface dont elles découlent.

En résumé, une interface est une classe abstraite sans donnée où toutes les méthodes sont abstraites, sans source associé.

Pour ce qui est de la définition des interfaces, il suffit de

- renseigner les méthodes associées, sans implémentation,
- faire précéder le nom de la classe par le mot réservé `interface`.

Ainsi, nous pouvons écrire

```
interface etudiant
{
    void Afficher();
}
```

et la classe abstraite `etud` définie en 0 peut dériver de `etudiant` moyennant la modification

```
abstract class etud : etudiant
```

Si l'héritage classique est impossible sous une forme multiple, il est autorisé dans le cas d'interfaces. Ainsi, nous pouvons écrire

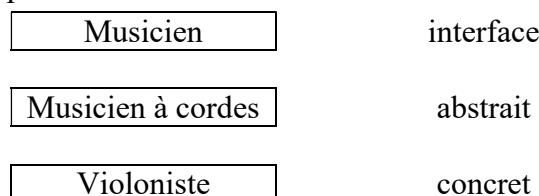
```
interface iA
{
    void fa1();
}
interface iB
{
    string fb();
}
```

et, en suivant les instructions affichées lors de la dactylographie de la première ligne du code suivant (touche Tabulation), nous obtenons

```
class derive : iA, iB
{
    #region Membres de iA
    public void fa1();
    {
        // TODO : ajoutez l'implémentation de derive.fa1
    }
    public void fa2(int a);
    {
        // TODO : ajoutez l'implémentation de derive.fa2
    }
    #endregion
    #region Membres de iB
    public string fb();
    {
        // TODO : ajoutez l'implémentation de derive.fb
        return null;
    }
    #endregion
}
```

et il ne nous reste plus qu'à compléter ces définitions minimalistes ...

Pour illustrer les liens entre les classes, les classes abstraites et les interfaces de manière moins informatique, nous pouvons considérer le schéma



Rapporté au monde de la programmation, nous définissons dans la classe *Musicien* les opérations que tout musicien doit être capable de faire :

- accorder son instrument,
- lire une partition,
- jouer de son instrument,
- ...

Dans la classe *Musicien à cordes*, nous implémentons certaines méthodes et complétons éventuellement :

- accorder son instrument (implémentation),
- lire une partition (implémentation),
- pincer les cordes (ajouter),
- tenir son archet (ajouter),
- ...

Enfin, dans la classe qui est visible par tous, à savoir *Violoniste*, nous pouvons implémenter tout ce qui est propre au violoniste lui-même :

- jouer de son instrument (implémentation),
- ...

Terminons par quelques conseils généraux prodigués par des développeurs professionnels (Microsoft, Borland, Sun) :

- les interfaces bien conçues sont plutôt petites et indépendantes les unes des autres,
- un trop grand nombre de fonctions rend l'interface peu maniable,
- une nouvelle interface sera créée lorsqu'une modification s'avère nécessaire,
- si la fonctionnalité créée peut être utile à des objets différents, une interface est à mettre en place,
- les interfaces sont adaptées à l'implémentation de fonctionnalités sous la forme de petits morceaux concis,
- les interfaces permettent de séparer l'utilisation et la conception des classes,
- deux classes peuvent partager la même interface sans descendre de la même classe mère.

7.4. La classe de base `object`

La classe `object` est le point de départ de toute l'architecture C# car toutes les classes en découlent. Ce mot réservé est en fait synonyme de la classe `System.Object`²⁸. Les méthodes de cette classe sont

- `Object()` est le constructeur,
- `Equals()` renvoie `true` si l'objet sur lequel porte la méthode est identique à celui transmis en argument, `false` sinon,
- `GetType()` renvoie le type de l'objet,
- `ToString()` est une fonction virtuelle (habituellement redéfinie) qui renvoie la chaîne correspondant à l'objet (`System.Object` en l'occurrence).

²⁸ Comme nous l'avons vu en 4.2, le mot réservé `object` n'est rien d'autre qu'un alias pour la classe `System.Object`.

Pour reprendre le cas de la méthode `ToString()`, nous pouvons l'adapter à la classe `etud` définie auparavant. Nous avons

```
class etud
{
    // Données membres et méthodes
    public override string ToString()
    {
        return "Eleve "+this.nom+" a eu "+this.cote+"/20";
    }
}
```

7.5. Manipulations d'objets

7.5.1. La recopie

Pour rappel, nous manipulons les objets par référence. Ce mécanisme est similaire à celui des pointeurs au point que, avec les définitions précédentes, les instructions

```
etud3TI ela = new etud3TI("Jones",19,"CliSer","ULg");
etud3TI elb = ela;
```

provoquent la création d'un objet renseigné par `ela` et `elb` car l'assignation porte sur la référence de `ela` et non le contenu. Nous avons abordé ce problème en 7.1.2.

Par conséquent, si nous effectuons l'instruction

```
elb.cote++;
```

nous transformons également la cote référencée par `ela` en 16.

Si nous souhaitons une véritable instruction de copie, il suffit que la classe `etud` implémente l'interface `ICloneable` et mette en place une méthode `Clone()`. Nous transformons alors notre définition de la classe `etud` en

```
class etud3TI : etud,ICloneable
{
    // Données membres et méthodes
    public object Clone()
    {
        return new etud(this.nom,this.Cote,this.option,this.stage);
    }
}
```

et la copie *réelle* qui permet de disposer de deux fiches distinctes (contenant les mêmes renseignements, dans un premier temps en tout cas) est effectuée par

```
etud3TI elb = (etud3TI) ela.Clone();
```

7.5.2. La comparaison

Comme indiqué en 7.1.2, nous pouvons aller plus loin et demander la comparaison d'objets. Nous devons alors l'implémenter comme pour la copie.

En effet, le code

```
etud eleveA = new etud();
eleveB = new etud();
if(eleveB == eleveA) Console.WriteLine("Etudiants identiques");
else Console.WriteLine("Etudiants différents");
```

n'effectue pas de comparaison sur le contenu de `ela` et `elb` mais bien sur leur emplacement en mémoire. Dans ce cas de figure, les étudiants seront différents puisqu'ils font référence à deux emplacements mémoire distincts alors que les contenus (vide) sont identiques.

Pour utiliser différemment les opérateurs de comparaison `==` et `!=`, il faut

- redéfinir la méthode `Equals()` de la classe `Object`,
- redéfinir les opérateurs `==` et `!=`,
- redéfinir la méthode `GetHashCode()` de la classe `Object`, cette méthode étant à la base de la comparaison d'objets.

Profitons pour signaler que les autres opérateurs (`+`, `-`, `...`) peuvent également être redéfinis²⁹. Les règles à suivre sont identiques à celles présentées ci-après, la méthode étant de type `static`.

Le code devient

```
class etud
{
    // Donnees membres et méthodes
    public override bool Equals(object obj)
    {
        return (this.nom == ((etud) obj).nom);
    }
    public static bool operator == (etud ea, etud eb)
    {
        return ea.Equals(eb);
    }
    public static bool operator != (etud ea, etud eb)
    {
        return ! ea.Equals(eb);
    }
    public override int GetHashCode()
    {
        return this.ToString().GetHashCode();
    }
}
```

7.6. Les structures dynamiques

Nous avons vu, en 6.4, que l'usage des pointeurs est déconseillé (`unsafe`). Cependant, les accrocs des structures dynamiques comme les listes, piles, files et autres arbres peuvent encore s'adonner à leur passe-temps favori.

En effet, le fait de disposer des classes et puisque ces dernières sont manipulées par référence selon leur adresse sur la pile, il est facile de simuler le traditionnel adressage par pointeurs cher aux adeptes du C.

Penchons-nous sur la simple liste. Pour ce faire, nous définissons la classe

```
class etud
{
    string nom;
    string prenom;
    etud suivant;
}
```

où nous renseignons les caractéristiques des étudiants et nous assurons le lien avec le suivant dans la liste par la simple déclaration

```
etud suivant;
```

²⁹ Le lecteur intéressé trouvera en annexe 10.2 un exemple complet portant sur la surcharge des opérateurs habituels. Il concerne des nombres rationnels (quotient de deux nombres entiers).

Supposons maintenant créer deux objets dérivant de cette classe `etud`. Il nous suffit d'écrire le code

```

• etud eleveA = new etud();
• eleveA.nom = "Winch";
• eleveA.prenom = "Largo";
• etud eleveB = new etud();
• eleveB.nom = "Blueberry";
• eleveB.prenom = "Mike";

```

et nous disposons de deux étudiants tout neufs `eleveA` et `eleveB`. Nous plaçons l'objet `eleveB` à la suite de `eleveA` par la seule instruction

```

• eleveA.suivant = eleveB;

```

7.7. Les définitions partielles de classes

Au fur et à mesure de l'évolution de notre travail, nous pouvons voir le code source associé à une classe prendre des proportions imposantes. Depuis la version 2.0, le langage C# nous permet de séparer la définition d'une classe sur plusieurs fichiers. Cette fonctionnalité est rendue possible par l'intermédiaire du mot `partial` qu'il suffit de placer avant chaque mot clé `class` concerné.

En nous basant sur ce qui précède, nous pourrions envisager de sauvegarder dans le fichier `etudiant_def.cs` les instructions

```

• partial class etud
• {public string nom;
•   public int cote;
•   public etud(string n,int c)
•   {nom = n;
•     cote = c;
•   }
•   public etud(int c,string n) : this(n,c)
•   {
•   }
•   public etud() : this("Inconnu",0)
•   {
•   }
• }

```

tandis que dans `etudiant_acces.cs`, nous plaçons

```

• partial class etud
• {public int Cote
•   {get
•     {return this.cote;
•   }
•   set
•     {if(value<0) this.cote=0;
•       else if(value>20) this.cote=20;
•       else this.cote=value;
•     }
•   }
• }

```

Signalons d'ailleurs que, dans le cadre de la création d'applications de type Windows, l'environnement de développement associe à toute fenêtre deux fichiers. Si nous baptisons une fenêtre `MyWindow`, nous découvrons deux fichiers :

- `MyWindow.designer.cs` qui reprend les données et méthodes générées automatiquement par Visual Studio en réponse à notre *design*,
- `MyWindow.cs` qui enregistre nos données et méthodes.

Par la suite, la version 3 du langage C# a introduit la notion de méthode qualifiées de partielles et renseignées également par le mot clé `partial`. Elles sont évidemment liées avec la notion de classe partielle que nous venons de découvrir.

Il s'agit de permettre aux programmeurs de déclarer la méthode dans un des fichiers associés à la classe et d'en définir les instructions associées dans un autre fichier. Cela permet notamment de compiler certaines parties de code sans savoir exactement ce que fait la fonction³⁰. Le travail en équipe s'en trouve ainsi facilité. Notons également que l'utilisation de certains générateurs automatiques de code permet de disposer de telles méthodes sans devoir se tracasser immédiatement de leur implémentation.

Les règles syntaxiques pour définir de telles méthodes sont les suivantes

- étant automatiquement définie en tant que `private`, ne pas renseigner de niveau de visibilité,
- être déclarée en tant que `partial`.

En reprenant le code précédent, nous pourrions ajouter, dans `etudiant_def.cs`,

```
partial string afficher();
```

et, dans `etudiant_acces.cs`,

```
partial string afficher()
{
    string rep = "";
    if (cote < 10)
        rep = nom + " est en échec (" + cote.ToString());
    else
        rep = nom + " a réussi (" + cote.ToString());
    return rep;
}
```

7.8. Les classes génériques

Le principe des classes génériques, comme le nom l'indique, est de définir des classes qui permettent d'appliquer des opérations à un ensemble d'autres classes.

7.8.1. Introduction

Nous avons rencontré, en algorithmique, le concept de liste linéaire. Ce concept permettait de définir, à partir de n'importe quel type de données, une *organisation* permettant de gérer (consulter, ajouter, supprimer, ...) un nombre d'éléments pouvant évoluer en cours d'exécution. En C#, cette organisation est accessible via la classe `List<>` où il suffit de renseigner, entre les symboles `<` et `>` le type d'objet que nous souhaitons manipuler. Par exemple, nous pouvons écrire *indifféremment*, en se basant sur la définition de 7.7,

```
List<int> ListeEntiers = new List<int>();
List<string> ListeChainesCaracteres = new List<string>();
List<Etud> ListeEtudiants = new List<Etud>();
```

Les différentes opérations disponibles dans le cadre des listes sont alors accessibles à la classe renseignée. Evidemment, le code associé à la classe générique se doit d'être adaptable à n'importe quelle classe.

³⁰ Nous pourrions comparer cette manière de faire avec la définition des prototypes en C.

Il s'agit d'un *template* (modèle) de comportements applicables à diverses situations, sur divers objets.

Généralement, une classe générique s'obtient à partir d'une classe existante. Partant des définitions suivantes de classes

```
class BaseClassique { }
class BaseGenerique<T> { }
```

nous pouvons créer des classes génériques de type, respectivement, *concret*, *construit fermé* ou *construit ouvert* selon le type de classe dont elles héritent.

```
class DeriveConcrete<T> : BaseClassique { }
class DeriveConstruitFerme<T> : BaseGenerique<int> { }
class DeriveConstruitOuvert<T> : BaseGenerique<T> { }
```

La différence entre les classes construites est que le type sur lequel s'appuie *BaseGenerique* est spécifié ou non.

7.8.2. Implémenter une classe générique par héritage

Comme exemple, nous pouvons nous pencher sur l'implémentation de la gestion d'une pile au sens algorithmique. Pour rappel, la définition, dans ce contexte, est la suivante :

Une pile est un ensemble formé d'un nombre variable, éventuellement nul, de données d'un même type sur lequel sont définies les opérations

- créer une pile vide,
- tester si la pile est vide,
- ajouter, en *dessus* de pile, une nouvelle donnée,
- supprimer la dernière donnée empilée si la pile n'est pas vide,
- consulter la dernière donnée empilée si la pile n'est pas vide.

Le type de données n'est, a priori, pas connu. Comme on peut le constater dans tout bon ouvrage traitant de l'algorithmique, la notion de pile est une simplification de celle de liste. Nous allons donc tout naturellement nous appuyer sur la définition existante de *List<>*. Une fois cette décision prise, il nous suffit, pour créer cette classe de gestion de piles, d'utiliser les différentes ressources associées à la classe mère. Nous pouvons écrire

```
public class PileGenerique<ClasseTraitee> : List<ClasseTraitee>
{
    public PileGenerique() : base()
    { }
    public bool EstVide
    { get { return Count == 0; } }
    public int Combien
    { get { return Count; } }
    public void Ajouter(ClasseTraitee ObjetTraite)
    { Insert(0, ObjetTraite); }
    public void Supprimer()
    { RemoveAt(0); }
    public ClasseTraitee Consulter()
    { return this[0]; }
}
```

Passons maintenant à un cas plus complexe d'une classe dont les membres ne sont pas définis a priori. L'utilisation de membres génériques pour cette classe semble donc appropriée mais nous allons nous rendre compte que certaines difficultés se présentent suite à cette approche.

7.8.3. Implémenter une classe générique par ses membres

Prenons l'exemple de véhicules automobiles réduits aux caractéristiques suivantes : le châssis, le moteur et les roues. Chaque voiture étant constituée d'éléments de ce type, il peut être intéressant de ne pas écrire un code associé à chaque *association* mais plutôt de créer un modèle qui permet *génériquement* chaque association.

A côté de ces données, nous ajoutons les méthodes rouler et arrêter.

Nous pouvons maintenant écrire, en première approche, notre classe de la manière suivante

```
public class Voiture<ClasseChassis, ClasseMoteur, ClasseRoues>
{
    private ClasseChassis chassis;
    private ClasseMoteur moteur;
    private ClasseRoues roues;
    public Voiture()
    {
    }
    public string Rouler()
    {
        return "Rouler à implémenter";
    }
    public string Arrêter()
    {
        return "Arrêter à implémenter";
    }
}
```

Le problème qui se pose est l'ignorance de la classe `Voiture` sur les classes utilisées en tant que données. Nous y sommes directement confrontés puisque le constructeur est destiné à instancier les différents champs de la classe.

Pour renseigner la classe hôte, nous devons renseigner les contraintes de type qui sont des renseignements sur les classes utilisées à l'*aveugle*. Ces contraintes de type sont déclarées par `where`. Dans ce cas de figure, nous indiquons que les classes utilisées disposent d'un constructeur par défaut. Dès lors, le compilateur peut effectuer son travail lors de l'instanciation des objets génériques. La classe `Voiture` devient alors

```
public class Voiture<ClasseChassis, ClasseMoteur, ClasseRoues>
{
    where ClasseChassis : new()
    where ClasseMoteur : new()
    where ClasseRoues : new()
    private ClasseChassis chassis;
    private ClasseMoteur moteur;
    private ClasseRoues roues;
    public Voiture()
    {
        chassis = new ClasseChassis();
        moteur = new ClasseMoteur();
        roues = new ClasseRoues();
    }
    public string Rouler()
    {
        return "Rouler à implémenter";
    }
    public string Arrêter()
    {
        return "Arrêter à implémenter";
    }
}
```

Pour aller plus loin dans ces contraintes de type et définir un canevas pouvant se révéler utile dans la définition des classes pouvant intervenir en tant que membres de la classe `Voiture`, nous pouvons compléter ce cadre de travail en utilisant des interfaces.

Par exemple, en nous concentrant sur le seul moteur, nous pouvons définir

```
public interface IMoteur
{
    int Cylindree { get; set; }
    int Puissance { get; set; }
    int Rapport { get; set; }
    string ChangerRapport(bool Monter);
}
```

A partir de ce moment, nous pouvons écrire

```
public class Voiture<ClasseChassis, ClasseMoteur, ClasseRoues>
{
    where ClasseChassis : new()
    where ClasseMoteur : IMoteur, new()
    where ClasseRoues : new()
}
```

Une fois ces ressources connues par `Voiture`, nous pouvons, par exemple, de manière sommaire, réécrire le code de la méthode `Rouler` pour le mettre sous la forme suivante.

```
public string Rouler()
{
    if(moteur.Rapport>0) return "Véhicule en mouvement";
    else return "Véhicule à l'arrêt";
}
```

Pour utiliser cette classe générique, nous pouvons agir comme nous l'avons fait avec `List<>`. Ainsi, la déclaration d'un objet, en supposant les classes `ChassisFamiliale`, `MoteurMoyen` et `RouesJantes` connues, se fait par une instruction du type

```
Voiture<ChassisFamiliale, MoteurMoyen, RouesJantes> unevoiture;
```

tandis que l'héritage peut prendre la forme

```
public class UneVoiture : Voiture<ChassisFamiliale, MoteurMoyen, RouesJantes>
{
}
```

7.8.4. Mixer classe et méthodes génériques

Lorsqu'une classe est générique, accéder à un paramètre de type renseigné au niveau de la classe de la manière suivante.

```
public class ClasseGenerique<T>
{
    void Methode(T Argument)
    { }
}
```

Cette manière de faire doit être adaptée lorsque la classe et la méthode sont génériques. En effet, même si le type est identique, il faut modifier le paramètre pour éviter que celui de la méthode ne puisse masquer celui de la classe (variable locale contre variable globale). Ainsi, on écrira le code suivant.

```
public class ClasseGenerique<T>
{
    void Methode<U>()
    { }
}
```

Tout comme pour les classes génériques, il est possible de compléter les définitions des méthodes génériques par des contraintes de type. Dans l'exemple suivant, nous supposons que l'argument transmis sera d'un type qui implémente la *comparaison*.

```
void PermuterSiPlusGrand<T>(ref T Arg1, ref T Arg2) where T : System.IComparable<T>
{
    T tmp;
    if (Arg1.CompareTo(Arg2) > 0)
    {
        tmp = Arg1;
        Arg1 = Arg2;
        Arg2 = tmp;
    }
}
```

La surcharge reste possible sur ce type de méthode et nous pouvons rencontrer le code suivant.

```
void Methode()
{ }
void Methode<T>()
{ }
void Methode<T, U>()
{ }
```

7.9. Les méthodes d'extension

Les méthodes d'extension ont été introduites avec la troisième version de C# et permettent d'étendre aisément les fonctionnalités disponibles pour une classe en y ajoutant des nouvelles méthodes sans devoir passer par une classe dérivée.

Ces méthodes d'extension seront `static` et le type du premier paramètre de la fonction sera précédé du mot clé `this` et fera référence à un objet de la classe *étendue*.

Pour illustrer ces propos quelque peu confus, passons par un exemple.

Supposons que nous souhaitions créer une méthode qui s'applique à une chaîne de caractères et renvoie son équivalent en majuscules. Quoique équivalente à la méthode déjà existante `ToUpperCase()`, nous la baptiserons `EnMajuscules()`.

Conformément à ce qui précède, nous pouvons écrire

```
public static class StringExtension
{
    public static string EnMajuscules(this string sChaine)
    { return sChaine.ToUpper(); }
}
```

et l'utiliser via l'instruction

```
Console.WriteLine("Hello World !".EnMajuscules());
```

Cette possibilité est bien pratique lorsque nous disposons d'une classe dont nous souhaitons étendre les fonctionnalités mais que cette classe se trouve dans une librairie pour laquelle nous ne disposons pas des sources.

8. Linq

8.1. Introduction

Linq est le diminutif de Language Integrated Query. Cela signifie, à la louche, que Linq est un langage d'interrogation intégré. Pour être plus précis, son intégration est faite par le langage C# à partir de la version 3.0³¹. Nous pouvons donc utiliser les ressources qui sont les siennes comme les ressources des classes fournies par le .Net Framework.

Dévoilons un coin du mystère entourant cet addendum au langage C# en signalant que, conformément à ce qui existe sur le marché en termes d'élaboration de requêtes, Linq est largement inspiré de la syntaxe SQL. Il l'associe en fait à la philosophie objets en permettant de construire des requêtes de type SQL à toute collection d'objets que l'on peut rencontrer dans C#.

Linq est une étape supplémentaire vers l'intégration des données à la philosophie objets. En effet, par l'intermédiaire de cette ressource d'interrogation, le langage C# peut manipuler des données en provenance de n'importe quel SGBD, de sources XML et de liste d'objets rendus disponibles par les diverses classes du .Net Framework. Pour placer la cerise sur le gâteau, nous terminerons en précisant que, quelle que soit la source de données, la manipulation est identique.

Nous pouvons maintenant nous pencher sur les diverses approches possibles dont nous disposons pour utiliser Linq.

8.2. Exemple introductif

Linq est un puissant langage de requête grâce à de nombreuses options dans sa syntaxe. Avant de présenter en détail ces multiples possibilités, nous allons nous contenter de présenter simplement le strict nécessaire.

Pour ce faire, nous pouvons récupérer une collection d'objets fournis par le système par l'intermédiaire d'une classe fournie par .Net.

Ainsi, nous récupérerons les caractéristiques de fichiers faisant partie d'un répertoire déterminé. Dans l'exemple ci-dessous, nous travaillons dans le répertoire `c:\program files\`. La classe `DirectoryInfo`, définie dans l'espace de noms `System.IO`, nous permet de nous positionner dans ce répertoire et d'en récupérer les caractéristiques.

Parmi les possibilités diverses et variées, nous allons utiliser la méthode `GetDirectories()` qui, comme son nom l'indique, fournit les différents répertoires contenus dans le répertoire de départ et leurs caractéristiques.

31 Linq est également intégré à Visual Basic sous .Net.

Nous pouvons écrire le code suivant

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

namespace LinqConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            DirectoryInfo info = new DirectoryInfo(@"c:\program files\");
            var query = from d in info.GetDirectories()
                        where d.Name.Contains("Microsoft")
                        orderby d.CreationTime ascending
                        select new { d.LastWriteTime, d.Name };
            foreach (object vv in query)
            {
                Console.WriteLine(vv.ToString());
                Console.ReadLine();
            }
        }
    }
}
```

Nous pouvons maintenant décortiquer la commande de requête en constatant que, conformément à ce qui a été annoncé, nous retrouvons un certain fumet de syntaxe SQL. La différence, c'est essentiellement l'ordre³² des options.

Nous avons

- `var` indique la définition d'une requête de type Linq, ce type étant utilisé dans la syntaxe Linq car il n'est pas possible de savoir, avant exécution, ce qui va être récupéré,
- `query` est le nom de la variable stockant le résultat de la requête,
- `from` permet de définir l'origine des données traitées,
- `where` permet de déterminer le filtre sur les données,
- `orderby`, comme on peut s'en douter, trie les données sur les champs indiqués,
- `select new` termine l'instruction en spécifiant ce qui est extrait comme données, les colonnes choisies étant incluses tandis que `select d` aurait permis de récupérer toutes les données fournies par `GetDirectories()`.

8.3. Linq et les collections d'objets

Pour illustrer les possibilités de Linq, nous allons nous appuyer sur un exemple simple d'objets ou, plus précisément, de collections d'objets.

Par exemple, nous considérons des personnes et des véhicules ainsi que les liens entre ces deux entités de données.

Nous commençons par définir les classes destinées à contenir les données puis passons en revue les méthodes permettant de créer ces dernières.

³² Linq est intégré à Visual Studio qui possède la facilité à la frappe du code : l'*Intellisense*. Pour en assurer le fonctionnement, il est nécessaire de commencer par lui fournir ce sur quoi il doit travailler. La place de la clause `from` devient dès lors naturelle.

Les fiches de personnes sont caractérisées par la classe suivante

```
class Personne
{
    public int IDPers { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public DateTime Naissance { get; set; }
    public Personne()
    {
    }
    public Personne(string Nom_, string Prenom_, DateTime Naissance_)
    {
        Nom = Nom_;
        Prenom = Prenom_;
        Naissance = Naissance_;
    }
    public Personne(int IDPers_, string Nom_, string Prenom_, DateTime Naissance_)
    : this(Nom_, Prenom_, Naissance_)
    {
        IDPers = IDPers_;
    }
}
```

Les véhicules, quant à eux, sont définis comme suit

```
class Voiture
{
    public int IDAuto { get; set; }
    public string Marque { get; set; }
    public string Serie { get; set; }
    public int Cyllindree { get; set; }
    public Voiture()
    {
    }
    public Voiture(string Marque_, string Serie_, int Cyllindree_)
    {
        Marque = Marque_;
        Serie = Serie_;
        Cyllindree = Cyllindree_;
    }
    public Voiture(int IDAuto_, string Marque_, string Serie_, int Cyllindree_)
    : this(Marque_, Serie_, Cyllindree_)
    {
        IDAuto = IDAuto_;
    }
}
```

Les personnes et les véhicules sont éventuellement associés par la classe de liens suivante

```
class Lien
{
    public int IDLien { get; set; }
    public int IDPersonne { get; set; }
    public int IDAuto { get; set; }
    public Lien()
    {
    }
    public Lien(int IDPersonne_, int IDAuto_)
    {
        IDPersonne = IDPersonne_;
        IDAuto = IDAuto_;
    }
    public Lien(int IDLien_, int IDPersonne_, int IDAuto_)
    : this(IDPersonne_, IDAuto_)
    {
        IDLien = IDLien_;
    }
}
```

Pour gérer ces classes, nous allons définir des objets, en statique, dans la classe principale qui est **Program** puisque nous pouvons travailler en mode console, de la manière suivante

```
static List<Personne> pers = new List<Personne>();
static List<Voiture> voiture = new List<Voiture>();
static List<Lien> lien = new List<Lien>();
```

Pour créer des personnes, la méthode suivante peut faire l'affaire.

```
static void RemplirPersonne()
{
    pers.Add(new Personne(1, "Winch", "Largo", new DateTime(1972, 08, 25)));
    pers.Add(new Personne(2, "Blackman", "Catherine", new DateTime(1976, 07, 18)));
    pers.Add(new Personne(3, "Wallenstein", "Georges", new DateTime(1970, 06, 19)));
    pers.Add(new Personne(4, "Cochrane", "Dwight", new DateTime(1978, 04, 22)));
    pers.Add(new Personne(5, "Carmichaël", "Lucie", new DateTime(1974, 01, 05)));
}
```


Ensuite viennent les données relatives aux voitures.

```
static void RemplirVoiture()
{
    voiture.Add(new Voiture(1, "Toyota", "Yaris", 1500));
    voiture.Add(new Voiture(2, "Toyota", "Corolla", 1600));
    voiture.Add(new Voiture(3, "Toyota", "Corolla", 1800));
    voiture.Add(new Voiture(4, "Toyota", "Corolla", 2000));
    voiture.Add(new Voiture(5, "Toyota", "Avensis", 1800));
    voiture.Add(new Voiture(6, "Toyota", "Avensis", 2000));
    voiture.Add(new Voiture(7, "Toyota", "Avensis", 2000));
}
```

Pour terminer, les liens entre les personnes et les voitures sont mis en place par la méthode suivante.

```
static void RemplirLien()
{
    lien.Add(new Lien(1, 1, 6));
    lien.Add(new Lien(2, 3, 7));
    lien.Add(new Lien(3, 4, 3));
    lien.Add(new Lien(4, 5, 1));
}
```

L'utilisation de ces données peut se faire dans la méthode `Main()` si nous travaillons en mode console. Nous pouvons écrire les lignes de code suivantes.

```
static void Main(string[] args)
{
    RemplirPersonne();
    RemplirVoiture();
    RemplirLien();
    RemplirAuto();
}
```

Nous pouvons maintenant, par l'exemple, examiner quelques possibilités offertes par Linq.

La première chose à faire est de rédiger une requête de consultation. Nous allons commencer par travailler sur une seule source de données. Par exemple, nous souhaitons récupérer la liste des personnes dont le nom contient la lettre `e`. Nous définissons alors une variable `q1` de type `var`.

- Les données se trouvent dans la variable `pers` définie auparavant. Cela entraîne l'apparition, en première position, de la clause `from p in pers`. La *variable* `p` renseigne l'entité à récupérer.
- Le filtre de recherche est défini par la contenance de la lettre `e` et cela se traduit par l'option suivante `where p.Nom.Contains('e')`
- Pour dérouter l'utilisateur, nous décidons de classer les entités par ordre alphabétique sur le prénom via l'option `orderby p.Prenom`
- Nous terminons cette instruction par l'opération à effectuer qui est une sélection en spécifiant ce que nous souhaitons récupérer, dans ce cas l'entité entière. `select p`

Une fois cette instruction exécutée, la variable `q1` contient une collection d'objets. Nous pouvons maintenant en consulter le résultat par l'intermédiaire d'une boucle de parcours. Les instructions suivantes permettent d'afficher, de manière *structurée*, l'identifiant, le nom et le prénom de chaque entité récupérée de type `Personne`.

```
foreach (Personne pp in q1)
    Console.WriteLine(pp.IDPers.ToString() + " : " + pp.Prenom + " " + pp.Nom);
```

La manipulation suivante consiste à ne pas se contenter des entités complètes mais de se limiter aux champs nécessaires. Tant qu'à faire, nous personnalisons également la récupération des champs nom et prénom en les concaténant dès la récupération.

Nous adaptons l'instruction précédente en introduisant le descriptif des enregistrements par l'intermédiaire du mot clé `new` qui permet de définir les différents champs à récupérer.

```
var q2 = from p in pers
        where p.Nom.Contains('e')
        orderby p.Prenom
        select new { p.IDPers, identification = p.Prenom + " " + p.Nom };
```

La boucle de parcours des données est similaire à celle qui précède et peut se mettre sous la forme suivante à la différence près que nous sommes limités aux champs récupérés ou définis.

```
foreach (var pp in q2)
    Console.WriteLine(pp.IDPers.ToString() + " : " + pp.identification);
```

Nous pouvons maintenant passer à l'étape suivante qui permet de mixer les données originaires de différentes sources. La jointure est au bout de cet exemple et, tant qu'à faire, nous allons considérer une jointure multiple reprenant les trois sources de données disponibles. Cette jointure est similaire à celle proposée par défaut en SQL par l'intermédiaire du mot clé `join`. Nous commençons par récupérer les données de la variable `pers` et les *joignons* avec celles de la variable `lien` pour terminer par celles de la variable `voiture`, les conditions portant sur les identifiants respectifs où nous remarquerons que l'opérateur `=` (ou `==`) n'est pas supporté, étant remplacé par `equals`. Pour terminer, nous ne conservons que les données `Marque`, `Serie` et `identification` définie comme précédemment. L'instruction correspondante peut se mettre sous la forme suivante.

```
var q3 = from p in pers
        join l in lien
        on p.IDPers equals l.IDPersonne
        join v in voiture
        on l.IDAuto equals v.IDAuto
        select new { identification = p.Nom + " " + p.Prenom, tmp.Marque, tmp.Serie,
                    tmp.Modele };
```

La consultation se fait par l'intermédiaire d'instructions similaires à celles rencontrées précédemment comme suit.

```
foreach (var aa in q3)
{
    Console.WriteLine(aa.identification);
    Console.WriteLine("- " + aa.Marque + " " + aa.Serie);
}
```

Notons toutefois qu'il existe plusieurs manières d'obtenir le même résultat. Nous pouvons commencer notre modification en mettant en évidence la jointure entre les variables `lien` et `voiture` qui, isolée, se mettrait sous la forme suivante.

```
from l in lien
join v in voiture
on l.IDAuto equals v.IDAuto
select new { l.IDPersonne, v.Marque, v.Serie }
```

Nous laissons au lecteur le soin d'aller plus avant dans la découverte de cette ressource proposée aux développeurs.

9. Les expressions régulières

9.1. Introduction

Les expressions régulières³³ ne sont pas propres au langage C#, loin de là. Elles sont couramment associées au langage Perl sous Unix.

Cependant, au vu de la puissance de traitement des chaînes de caractères qu'elles mettent à disposition du programmeur, .Net a intégré une classe permettant de manipuler de telles expressions. Comme il est possible de manipuler cette classe en C#, il devient alors naturel d'aborder ces notions dans le cadre de ce cours.

Les expressions régulières permettent de vérifier qu'une chaîne de caractères donnée respecte une certaine *forme*, une certaine *définition*. Les exemples qui suivent permettront au lecteur de se rendre compte de ce que signifie cette *forme* à respecter.

Elles sont utilisées pour

- vérifier la *forme* d'une chaîne de caractères,
- remplacer une partie de la chaîne de caractères respectant la *forme* définie par une autre,
- découper une chaîne de caractères.

En fait, une seule expression régulière permet d'éviter un nombre non négligeable d'instructions mettant en œuvre les fonctions s'appliquant aux chaînes de caractères comme, par exemple, `SubString()`, `IndexOf()`, `Replace()`, ... Evidemment, il y a des risques pour que, plus le code de départ est rendu compact, plus l'expression régulière qui s'y rapporte est complexe.

L'obtention d'une expression régulière efficace ne se fait pas sans peine. Nous voilà prévenus.

9.2. Syntaxe

Pour définir la forme d'une chaîne de caractères correspondant à une expression régulière, nous disposons de *métacaractères*. Hormis la complexité sous-jacente, nous pouvons comparer ces ressources à l'utilisation de métacaractères (`%d`, `%f`, ..., `\n`, `\t`, ...) dans la définition des chaînes de caractères destinées à afficher des résultats à l'écran par l'intermédiaire de la fonction `printf()` en C.

³³ Les expressions régulières trouvent leur origine dans la théorie des automates programmables et la théorie des langages formels dans le domaine de l'informatique (ensemble de chaînes de caractères). Les travaux des mathématiciens Warren Mc Culloch et Walter Pitts ont été utilisés par un autre mathématicien Stephen Kleene pour finalement déboucher sur une utilisation *industrielle* par Ken Thompson, un des fondateurs de Unix qui a entre autre mis en place `ged` (l'ancêtre de `ed`) et `grep`.

Le tableau suivant reprend les métacaractères sur lesquels s'appuie la construction des expressions régulières.

Symbole	Correspondance
\	Caractère d'échappement
^	Début de ligne
.	N'importe quel caractère
\$	Fin de ligne
	Alternative
()	Groupement
-	Intervalle de caractères
[]	Ensemble de caractères
[^]	Tout sauf un ensemble de caractères
+	Une fois ou plus
?	Zéro ou une fois
*	Zéro fois ou plus
{ x }	x fois exactement
{ x, }	Au moins x fois
{ x, y }	x fois minimum, y maximum

Illustrons par quelques exemples l'utilisation de ces métacaractères. L'expression régulière est définie dans la colonne de gauche, la deuxième colonne présente la recherche effectuée, la troisième colonne permet de visualiser la recherche effectuée par l'expression régulière sur différentes chaînes de caractères et la quatrième et dernière colonne renseigne simplement si la recherche est fructueuse ou non.

Expression	Recherche ...	Test	
^1	le chiffre 1 en début de chaîne	16548 99 561846	☺ ☹ ☹
a\$	le caractère a en fin de chaîne	Betaa Mu	☺ ☹
^.\$	un seul caractère, n'importe lequel		
^A\$	le seul caractère A		
a A	les caractères a ou A	Alpha Omicron	☺ ☹
A-Z	les lettres majuscules	M. Navratilova m. hingis	☺ ☹
[\ .]	le point	E. Merckx Eddy Merckx	☺ ☹
[0-9]	les chiffres	01/03 Premier mars	☺ ☹
[^ 0-9]	les symboles qui ne sont pas des chiffres	01/03 Premier mars	☺ ☹
^[^A]	autre chose que A en début de chaîne	Omicron Alpha	☺ ☹

0+ [1-9]	tout chiffre différent de 0 précédé d'au moins un 0	54 054 00054 00	☹ ☺ ☺ ☹
0? [1-9]	tout chiffre différent de 0 précédé d'au plus un 0	54 054 00054 00	☺ ☺ ☺ ☹
0* [1-9]	tout chiffre différent de 0 précédé ou non de 0	54 054 00054 00	☺ ☺ ☺ ☹
5{2}	toute séquence de deux 5 consécutifs	054 0554 05554 055554	☹ ☺ ☺ ☺
5{2,}	toute séquence d'au moins deux 5 consécutifs	054 0554 05554 055554	☹ ☺ ☺ ☺
5{2,4}	toute séquence de deux, trois ou quatre 5 consécutifs	054 0554 05554 055554 0555554	☹ ☺ ☺ ☺ ☺

A ces métacaractères de base, nous en ajoutons d'autres destinés à la *mise en forme* du texte.

Alias	Correspondance
\a	caractère de sonnerie
\n	caractère de nouvelle ligne
\r	caractère de retour à la ligne
\f	caractère de nouvelle page
\t	caractère de tabulation
\v	caractère de tabulation verticale

Nous disposons également de caractères qui permettent de synthétiser certains métacaractères de base dans le but de faciliter l'écriture et la lecture

(...)	définit un modèle, une sous-chaîne, avec mémorisation
\num	récupère un modèle enregistré, selon la valeur entière de <i>num</i>
\b	correspond à une limite (espace) représentant un mot
\B	correspond à une limite ne représentant pas un mot
\d	chiffre ([0-9])
\D	caractère différent d'un chiffre
\s	caractère d'espacement (espace, tabulation, saut de page, ...)
\S	caractère différent d'un caractère d'espacement
\w	caractère alphanumérique ou de soulignement ([a-zA-Z0-9_])
\W	caractère non alphanumérique ou de soulignement
\...	caractère codé en octal (\101 pour A)
\x...	caractère codé en hexadécimal (\x41 pour A)

La complexité des expressions régulières réside dans leur définition. C'est pourquoi nous allons commencer par présenter quelques expressions régulières qui ont le mérite d'être directement utilisables dans un contexte de programmation.

- **Balises HTML**

```
<([a-z][a-z0-9]*)\b[^\>]*>(.*)</\1>
```

La séquence `\b` permet de se limiter à ce qui précède un éventuel espace tandis que `\1` renseigne le premier modèle enregistré.

Dès lors, cette expression régulière permet de repérer toute occurrence d'une chaîne de caractères commençant par `<` suivi d'une lettre (minuscule) (`[a-z]`) puis, éventuellement, de lettres ou de chiffres (`[a-z0-9]`). La partie derrière `<` est alors *stockée* (`\b`). Viennent ensuite un caractère différent de `>` (`[^\>]`) et, éventuellement, *n'importe quoi* (`(.*)`). La chaîne de caractères cherchée se termine par la partie enregistrée (`\1`) entourée des symboles `<` et `>`.

- **Adresses IP**

Dire qu'une adresse IP est composée de 4 triplets de chiffres est un peu court puisque ces triplets doivent être compris entre 0 et 255.

Voyons d'abord comment écrire la définition d'un nombre compris entre 0 et 255. Nous avons les nombres 250 à 255 (`25[0-5]`), 200 à 249 (`2[0-4][0-9]`) et tous les autres compris entre 0 et 199, préfixés ou non par un ou deux 0 (`[01]?[0-9][0-9]?`).

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

Nous pouvons répéter cette séquence quatre fois en les séparant par un point obligatoire (`\.`). De la même manière, nous pouvons demander à une triple répétition de l'expression immédiatement suivie d'un point puis de la reprise de cette même expression. Pour terminer, nous demandons à ce que la chaîne soit limitée par un espace en tête de chaîne. Finalement, nous écrivons

```
\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

- **Nombre réel**

Un nombre réel est défini par un signe facultatif, un nombre entier puis une éventuelle virgule suivi d'un autre nombre entier. Pour terminer, nous accepterons un nombre dont la partie entière n'est pas renseignée, cela étant équivalent à une partie entière nulle (`.54` vaut `0.54`).

La première approche est de rechercher les chaînes composées d'un signe optionnel (`[-+]?`), d'une partie entière optionnelle (`[0-9]*`) suivie éventuellement d'un point (`\.?`) pour terminer par une partie décimale tout autant optionnelle (`[0-9]*`). Nous écrivons alors

```
[-+]?[0-9]*\.[0-9]*
```

Le problème de cette expression est que tout est optionnel et que, par conséquent, une chaîne vide peut faire l'affaire.

L'approche suivante est de considérer séparément les nombres avec parties décimales (`[-+]?[0-9]*\.[0-9]+`) et les autres qui sont alors entiers (`[0-9]+`). L'expression est alors la disjonction des deux précédentes, in extenso

```
[-+]?([0-9]*\.[0-9]+|[0-9]+)
```

Le lecteur courageux établira l'équivalence avec l'expression régulière suivante

```
[-+]?[0-9]*\.[0-9]+
```

- **Adresses eMail**

Une adresse eMail est constituée de trois parties, à savoir

- une partie locale qui identifie la *boîte* aux lettres électronique en tant que telle,
- le caractère @,
- un nom de domaine qui identifie l'hébergeur de la *boîte*.

Si on se limite aux adresses composées des caractères alphanumériques et des caractères spéciaux `_`, `-` et `.`, nous pouvons utiliser l'expression régulière suivante qui encadre l'adresse eMail par deux *espaces* et fait abstraction de la casse.

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

9.3. Les expressions régulières en C#

Tout d'abord, il faut signaler que cette ressource est accessible via l'espace de noms `System.Text.RegularExpressions`.

Ensuite, nous devons utiliser un objet de type `Regex` et pouvons transmettre l'expression régulière en tant que paramètre au constructeur. Un second argument peut être également renseigné et reprend les options souhaitées pour le traitement de l'expression régulière. Par exemple, `IgnoreCase` ne respecte pas la casse.

Nous commençons par vérifier si une expression régulière est valide ou non. Il suffit d'instancier un objet de type `Regex` en l'intégrant à un bloc `try - catch` de la manière suivante. Si l'instanciation est effective, l'expression régulière est valable et nous terminons le bloc `try` en renvoyant `true`. Sinon, nous passons dans le bloc `catch` où nous renvoyons `false` correspondant *heureusement* à une expression régulière invalide.

```
private bool Validite(string sExpressionReguliere)
{
    try
    {
        Regex test = new Regex(sExpressionReguliere);
        return true;
    }
    catch
    {
        return false;
    }
}
```

Pour vérifier la validité d'une expression régulière sur une chaîne de caractères, la méthode mise à disposition est `IsMatch()`. Elle renvoie une valeur booléenne. Ainsi, le code suivant met en place une méthode qui vérifie si la chaîne de caractères contient une adresse mail correcte, la prudence dictant d'intégrer à nouveau un bloc `try - catch`.

```
public bool VerifierMail(string sAVerifier)
{
    try
    {
        Regex exp = new Regex(@"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b",
                               RegexOptions.IgnoreCase);
        return exp.IsMatch(sAVerifier);
    }
    catch
    {
        return false;
    }
}
```

Nous pouvons évidemment généraliser cette fonction en ajoutant l'expression régulière et l'option comme paramètres. Nous obtenons

```
public bool Correspondance(string sExpressionReguliere,
                          string sAVerifier,
                          RegexOptions roParametres)
{
    try
    {
        Regex test = new Regex(sExpressionReguliere, roParametres);
        return test.IsMatch(sAVerifier);
    }
    catch
    {
        return false;
    }
}
```

En ce qui concerne les sous-chaînes correspondant éventuellement à l'expression régulière, il est possible de travailler *individuellement* ou *en groupe*. En effet, les méthodes `Match()` et `Matches()` de la classe `Regex` renvoient respectivement le premier élément trouvé (de type `Match`) ou tous les éléments correspondants (de type `MatchCollection`).

Les deux fonctions suivantes permettent de récupérer une ou plusieurs occurrences selon ce qui précède. Notons toutefois qu'il conviendrait de tenir compte de la validité de l'expression régulière `sExpressionReguliere` transmise.

```
private Match TrouverOccurrence(string sExpressionReguliere,
                                string sAVerifier,
                                RegexOptions roParametres)
{
    Regex test = new Regex(sExpressionReguliere, roParametres);
    return test.Match(sAVerifier);
}
private MatchCollection TrouverOccurrences(string sExpressionReguliere,
                                           string sAVerifier,
                                           RegexOptions roParametres)
{
    Regex test = new Regex(sExpressionReguliere, roParametres);
    return test.Matches(sAVerifier);
}
```

Pour se rendre compte de ce qui se passe effectivement, nous pouvons placer dans une fenêtre un composant de type `RichTextBox` qui contiendra le texte à *analyser*. La fonction `TrouverOccurrences` renverra alors la collection d'occurrences relevées dans le composant de type `RichTextBox`. Cette collection est destinée à être transmise à la fonction suivante qui, après une remise à plat de la couleur de fond, mettra les différentes occurrences en les surlignant, la variable `nPosition` permettant de retrouver l'endroit du curseur après traitement. Nous pouvons écrire

```
private void MettreEnEvidence(MatchCollection Occurences, RichTextBox Controle)
{
    Controle.Select(0, Controle.Text.Length);
    Controle.SelectionBackColor = Color.Transparent;
    foreach (Match Occurence in Occurences)
    {
        Controle.Select(Occurence.Index, Occurence.Length);
        Controle.SelectionBackColor = Color.Silver;
    }
}
```

Comme nous pouvons le constater, les propriétés `Index` et `Length` permettent de localiser les différentes occurrences. Nous y ajoutons la propriété `value` qui, pour sa part, permet de récupérer la valeur de l'occurrence.

Après la recherche et le traitement de cette dernière, nous pouvons aborder le remplacement pris en charge par la méthode `Replace()`. Partant du même schéma que précédemment, il nous suffit d'ajouter un argument de type `string` pour renseigner la chaîne de caractères devant remplacer les diverses occurrences, nous pouvons écrire

```
private string RemplacerOccurrence(string sExpressionReguliere,
                                   string sTexteDepart,
                                   string sRemplacement,
                                   RegexOptions roParametres)
{
    Regex test = new Regex(sExpressionReguliere, roParametres);
    return test.Replace(sTexteDepart, sRemplacement);
}
```

Ainsi, si nous souhaitons supprimer les espaces superflus dans une chaîne de caractères baptisée `sDepart`, il nous suffit d'écrire

```
string sTraite = RemplacerOccurrence(@"\s+", sDepart, " ", RegexOptions.None);
```

Signalons que la méthode `Replace()` admet un argument entier supplémentaire qui renseigne le nombre d'occurrences qui doivent être éventuellement remplacées.

Pour en terminer avec cette méthode de remplacement, il est bon de savoir qu'elle implémente l'enregistrement de parties d'expressions régulières comme nous l'avons vu en 9.2. Ainsi, il est possible, dans un texte HTML, de baliser automatiquement toute occurrence d'une adresse eMail. Pour ce faire, il faut référencer la partie enregistrée de l'expression régulière dans la partie remplaçante suivant l'index d'enregistrement précédé du symbole `$`. A partir d'une chaîne de caractères baptisée `sHTML`, nous utilisons la méthode `RemplacerOccurrence()` précédemment définie comme suit.

```
string sTraite = RemplacerOccurrence(@"\b[A-Z0-9._%+-]+\@[A-Z0-9.-]+\.[A-Z]{2,4}\b",
                                     sHTML,
                                     @"<a href=\"mailto:$1\">$1</a>",
                                     RegexOptions.IgnoreCase);
```

Nous pouvons terminer ce chapitre par la découverte de la méthode `Split()`. Comme son nom l'indique, elle permet de séparer une chaîne de départ en plusieurs sous-chaînes et est finalement équivalente à son homologue issu de la classe `string` si ce n'est que le séparateur peut être plus complexe qu'un simple caractère.

Pour illustrer l'utilisation de cette méthode, nous allons rédiger une méthode qui transforme une chaîne de caractères en un tableau de chaînes de caractères obtenues à partir d'un *découpage* de celle d'origine selon les nombre rencontrés.

```
public string[] DecouperOccurrence(string sExpressionReguliere,
                                   string sATraiter,
                                   RegexOptions roParametres)
{
    Regex exp = new Regex(sExpressionReguliere, roParametres);
    return exp.Split(sATraiter);
}
```

Signalons, pour terminer ce chapitre, que les méthodes vues sont également définies en tant que méthodes statiques et qu'il n'est donc pas obligatoire de passer par l'instanciation d'un objet de type `Regex`, l'expression régulière faisant partie des arguments à transmettre à la méthode. Par exemple, les instructions de la dernière méthode peuvent se mettre sous la forme

```
return Regex.Split(sATraiter, sExpressionReguliere, roParametres);
```

10. Annexes

10.1. Premier contact avec la P.O.O. : Programmation graphique sous GDI+

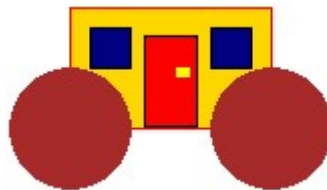
Nous abordons, dans cette annexe, un programme utilisant les ressources graphiques de base de .NET. Ce programme nous permet d'illustrer la programmation orientée objet et ses classes et sert de support au laboratoire de programmation en P.O.O.

Dans Windows XP, la gestion des graphiques vectoriels 2D se fait par l'intermédiaire de GDI+ (implémentation avancée de l'interface graphique : *Graphics Design Interface*). Ce sous-ensemble du système d'exploitation est rendu accessible aux développeurs par l'intermédiaire de la plateforme .NET qui utilise un ensemble de classes : l'*interface de classes managées* vers GDI+. En tant que programmeurs, nous pouvons alors disposer de GDI+ pour les graphismes vectoriels à deux dimensions, les images et la typographie. Ces *créations artistiques* peuvent trouver un écho sur l'écran ou l'imprimante indépendamment du périphérique choisi, le système d'exploitation prenant en charge la conversion par l'intermédiaire des pilotes spécifiques.

Avant d'aborder l'analyse du problème à proprement parler, nous commencerons par présenter quelques manipulations sous Visual Studio pour disposer d'une fenêtre hôte de notre application puis aborderons brièvement quelques ressources définies en GDI+.

10.1.1. Présentation de l'application

Il s'agit d'illustrer les concepts orientés objet en construisant un carrosse ou, pour être plus exact, un engin qui pourrait ressembler à un carrosse. Notre chef d'œuvre graphique ressemble à

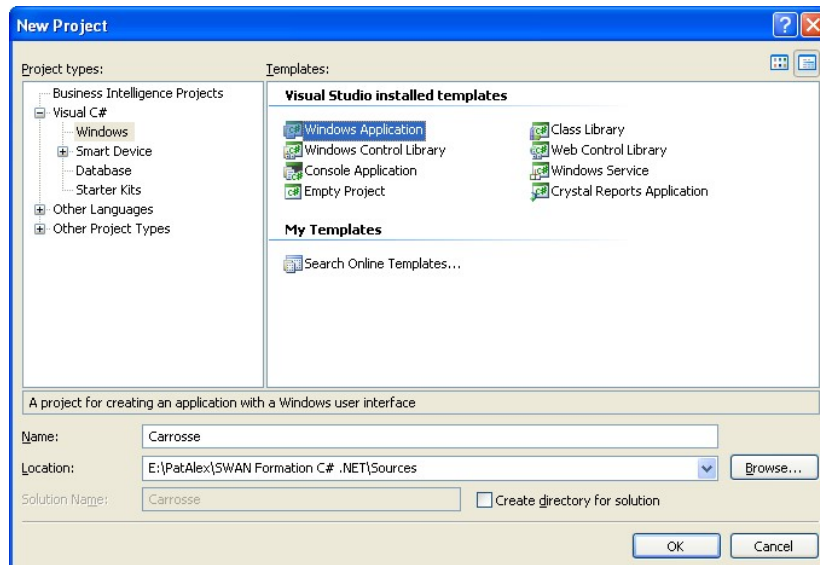


Ce carrosse sera alors capable de traverser l'écran de gauche à droite dès l'activation du mouvement effectuée par l'intermédiaire d'un bouton.

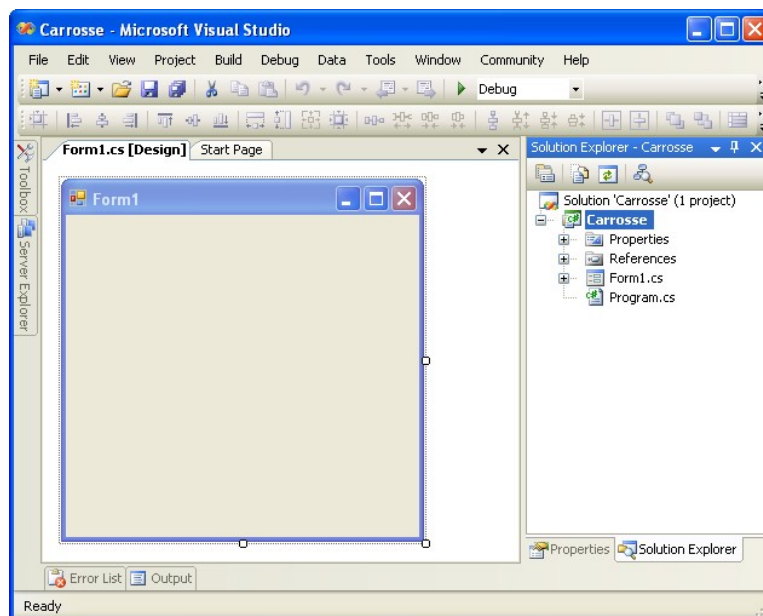
10.1.2. Ressources événementielles

La partie création et manipulation du carrosse est reprise en détail par la suite et illustre les concepts orientés objets que nous souhaitons développer. Cependant, pour visualiser le résultat de nos efforts, nous devons répondre à quelques exigences de l'environnement de développement permettant de dessiner sur une fenêtre de type Windows.

Pour commencer, nous pouvons créer une application de type fenêtrée. Il nous suffit, sous Visual Studio, de sélectionner le point de menu **File, New, Project ...** pour découvrir la fenêtre suivante, où les options proposées nous conviennent parfaitement.



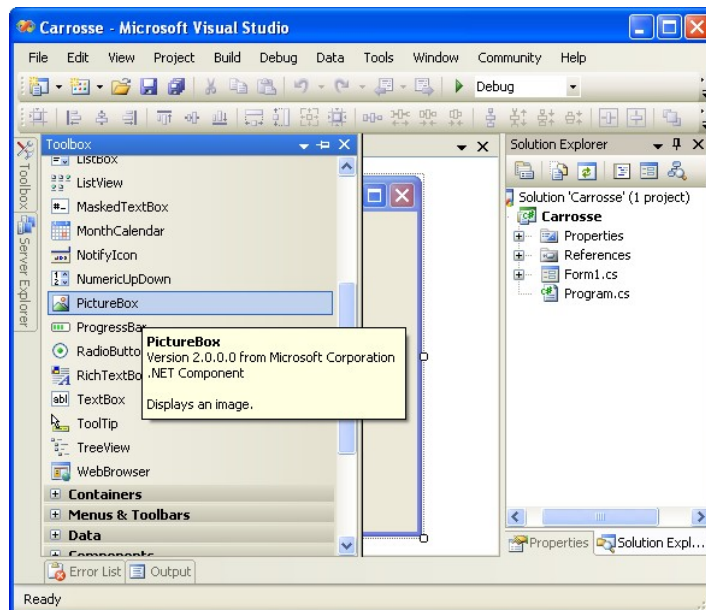
Nous nous retrouvons alors confrontés à un écran semblable à celui-ci.



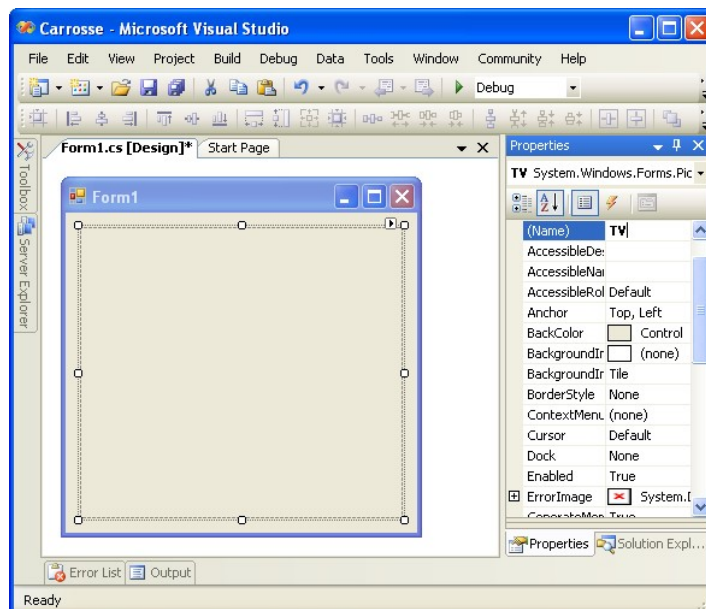
Nous apprendrons plus tard à personnaliser les éléments d'une telle fenêtre et nous limitons actuellement au strict nécessaire.

En fait, l'affichage se fera dans une zone particulière de la fenêtre délimitée par un composant de type `PictureBox` que nous pouvons trouver dans la **Toolbox**.

Une fois sélectionné, nous faisons glisser ce composant sur notre fenêtre. Nous passons donc par l'étape



Nous personnalisons alors notre composant PictureBox en le déplaçant (manipulation souris), l'agrandissant (manipulation souris) et le baptisant (découverte de la fenêtre Properties). Ainsi, pour donner le nom TV à ce composant, nous passons par la manipulation (champ Name de la fenêtre Properties mis à TV).



Nous répétons approximativement les mêmes opérations pour placer un composant de type Button que nous baptiserons BtnCarrosse.

10.1.3. Ressources GDI+

L'espace de noms dédié aux traitements graphiques est `System.Drawing.Graphics`. On y trouve notamment la classe `Graphics` qui représente une surface de dessin GDI+ et est le point de départ à la création de représentations graphiques.

C'est ainsi que nous pouvons découvrir, au sein de l'application ou grâce à l'aide, que les objets dérivant de la classe `Graphics` disposent de méthodes telles que

- `DrawArc()` permet de dessiner un arc d'ellipse (origine en 0° et avancement dans le sens non trigonométrique, comme les aiguilles d'une montre) selon les paramètres transmis, par exemple,
 - un objet de type `Pen` renseigne le crayon utilisé pour dessiner le contour de l'arc d'ellipse (couleur et épaisseur),
 - `x` et `y` déterminent le coin supérieur gauche du rectangle *contenant* l'ellipse,
 - `width` et `height` fixent la longueur et la hauteur du rectangle *contenant* l'ellipse à dessiner.
- `startAngle` et `sweepAngle` renseignent les angles de début et de fin d'arc.
 - `FillRectangle()` permet de dessiner le périmètre d'un rectangle selon les paramètres transmis, par exemple,
- un objet de type `SolidBrush` renseigne le remplissage utilisé pour dessiner l'intérieur du rectangle (couleur),
- `x` et `y` déterminent le coin supérieur gauche du rectangle,
- `width` et `height` fixent la longueur et la hauteur du rectangle à dessiner.
 - `DrawString()` nous donne la possibilité de placer du texte sur un graphique selon les paramètres transmis, par exemple,
- une chaîne de caractères désignant le texte à afficher,
- la police de caractères à utiliser, par l'intermédiaire d'un objet de type `Font`, pour afficher le texte choisi, incluant la taille des caractères,
- le pinceau servant à dessiner le texte sous la forme d'un objet de type `Brush` (ou, pour être plus exact, un objet qui en dérive comme `SolidBrush`),
- la position selon les coordonnées du coin supérieur gauche du rectangle destiné à entourer le texte.

Il existe encore bien d'autres ressources. Il suffit, pour les découvrir, de se rendre dans la documentation de Visual Studio et d'examiner les méthodes associées à la classe `Graphics`.

Nous allons maintenant nous pencher sur la manière de faire apparaître ces dessins construits à partir des ressources GDI+.

10.1.4. Comment intégrer ces ressources GDI+

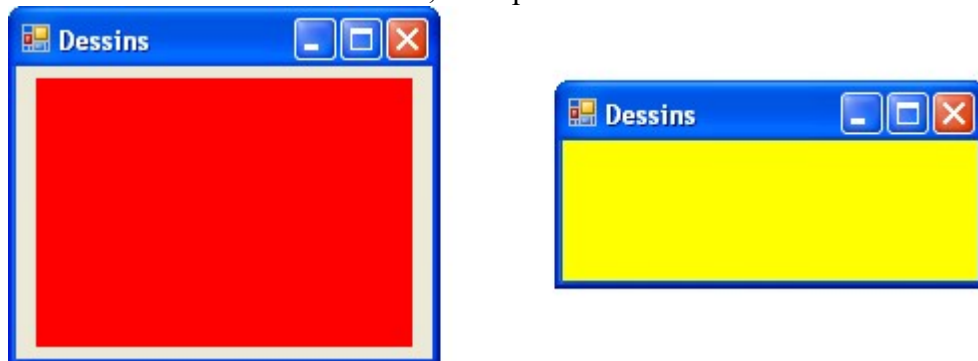
Il existe quatre manières d'utiliser des objets issus de cette classe. Nous distinguons

- La création à partir d'un paramètre `PaintEventArgs`

Lorsque, pour une raison ou une autre, un contrôle est appelé à être redessiné, un événement³⁴ est généré. Comme il est possible d'associer une méthode à un événement quelconque, nous pouvons, en cas de rafraîchissement de l'affichage du contrôle incriminé, associer une méthode particulière : `Paint()`. Ainsi, à partir de la fenêtre de travail, nous pouvons écrire

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    if (ClientSize.Width > 100 && ClientSize.Height > 50)
        e.Graphics.FillRectangle(new SolidBrush(Color.Red),
                                10, 6, ClientSize.Width - 20, ClientSize.Height - 12);
    else
        e.Graphics.FillRectangle(new SolidBrush(Color.Yellow),
                                0, 0, ClientSize.Width, ClientSize.Height);
}
```

Ainsi, selon les dimensions de la fenêtre, nous pouvons *admirer*



- La création à partir d'un *handle* d'élément

Cette manière de procéder vient du C++ et des API Windows et revient en fait à utiliser une sorte de pointeur vers le composant spécifié. Pour utiliser cette approche, nous devons créer un objet de type `Graphics` puis le relier au *handle* de la ressource utilisée comme support de dessin. Contrairement à la façon précédente, cette manière de faire n'est pas reliée à un événement particulier.

En reliant la méthode qui suit à l'événement `Click` correspondant à un clic de souris sur la fenêtre, nous pouvons écrire

```
private void Form1_Click(object sender, EventArgs e)
{
    Graphics gr = Graphics.FromHwnd(this.Handle);
    gr.FillEllipse(new SolidBrush(Color.Brown), 55, 25, 10, 10);
    gr.FillEllipse(new SolidBrush(Color.Brown), 135, 25, 10, 10);
    gr.DrawEllipse(new Pen(Color.Coral, 3), 50, 20, 20, 15);
    gr.DrawEllipse(new Pen(Color.Coral, 3), 130, 20, 20, 15);
    gr.DrawArc(new Pen(Color.Red, 5), 20, 40, 160, 80, 0, 180);
}
```

³⁴ Nous n'entrons pas dans les détails de cette approche car cela touche directement la programmation événementielle, *matière suivante*.

Néanmoins, pour que ce code soit efficace, il faut le relier avec l'événement `Paint` de la fenêtre. Nous disposons de deux méthodes :

- taper le code puis le relier à l'événement en choisissant la méthode définie par l'intermédiaire de l'onglet **Events** de la fenêtre **Properties**,
- double cliquer dans la case située à côté de l'événement `Paint` dans la fenêtre **Properties**, onglet **Events**.

Nous obtenons alors le résultat suivant, après avoir cliqué sur la fenêtre,



- La création à partir d'un objet image

Nous pouvons directement travailler sur un fichier graphique de type bitmap³⁵ et ajouter des formes propres à la classe `Graphics`. Pour ce faire, nous plaçons un composant de type `PictureBox` sur la fenêtre et renseignons un fichier d'extension `.bmp` comme `Image` (fenêtre `Properties`) pour visualiser directement les effets de nos instructions.

Nous associons alors les instructions suivantes à l'événement `Load` de la fenêtre qui se produit au chargement de cette dernière

```
private void Form1_Load(object sender, EventArgs e)
{
    Graphics gr;
    gr = Graphics.FromImage(pbFun.Image);
    gr.FillEllipse(new SolidBrush(Color.Yellow), 5, 5, 30, 30);
    gr.DrawString("Bonjour du Paradis", new Font("Comic Sans MS"),
        new SolidBrush(Color.Cyan), 90, 212);
}
```

Nous pourrions alors constater les dégâts de l'ajout d'un soleil et de notre message



- La création à partir de la méthode `FromHdc`

Nous ne étendrons pas sur cette manière de faire car elle est similaire à celle s'appuyant sur le *handle* d'objets, si ce n'est que l'objet de type `Graphics` est créé à partir d'un autre objet de même type.

En repartant de la méthode `Paint`, nous obtenons le résultat associé au *handle* avec le code

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    IntPtr hdc = e.Graphics.GetHdc();
    Graphics nouveaugr = Graphics.FromHdc(hdc);
    nouveaugr.FillEllipse(new SolidBrush(Color.Brown), 55, 25, 10, 10);
    nouveaugr.FillEllipse(new SolidBrush(Color.Brown), 135, 25, 10, 10);
    nouveaugr.DrawEllipse(new Pen(Color.Coral, 3), 50, 20, 20, 15);
    nouveaugr.DrawEllipse(new Pen(Color.Coral, 3), 130, 20, 20, 15);
    nouveaugr.DrawArc(new Pen(Color.Red, 5), 20, 40, 160, 80, 0, 180);
}
```

³⁵ Les objets **Graphics** ne peuvent être créés qu'à partir de fichiers `.bmp` non indexés (fichiers `.bmp` 16 bits, 24 bits et 32 bits). Alors, chaque pixel d'un tel fichier contient une couleur.

10.1.5. Description des classes : données

Le but de l'application est de déplacer sur l'écran un carrosse. Ce dernier est composé de deux roues, deux fenêtres, une porte et sa poignée sans oublier l'habitacle qui fait le lien entre les divers éléments mentionnés.

Si nous analysons les différents composants de ce carrosse, nous pouvons constater qu'il existe deux disques et cinq rectangles. Par conséquent, nous allons définir deux classes sur lesquelles pourra s'appuyer le carrosse : `MonRectangle` et `MonCercle`.

Nous allons considérer que l'habitacle est la base du carrosse sur laquelle viennent se positionner tous les autres éléments. C'est pour cette raison que la classe `Carrosse` hérite de la classe `MonRectangle`.

Nous pouvons alors présenter les données qui constituent la classe `Carrosse` :

```
public class Carrosse : MonRectangle
{
    private MonCercle RoueG, RoueD;
    private MonRectangle Porte, FenD, FenG, Poignee;
}
```

Il nous reste maintenant à définir les classes `MonCercle` et `MonRectangle`.

Ces dernières sont toutes deux caractérisées par un positionnement, à savoir un couple de coordonnées (x,y). Par contre, nous devons enregistrer la longueur et la largeur (hauteur) dans le cas du rectangle tandis que le rayon suffira pour le cercle.

Puisque ces deux classes nécessitent la définition d'un emplacement par l'intermédiaire de coordonnées (x,y), nous pouvons également créer une classe `MonPoint` possédant deux données entières x et y. Les classes `MonRectangle` ou `MonCercle` en héritent toutes deux.

En ce qui concerne les *coloriages*, les classes `MonRectangle` et `MonCercle` sont des objets dont nous pouvons colorier l'intérieur. Il est donc logique de prévoir une couleur de remplissage ainsi qu'une variable booléenne renseignant si le remplissage doit ou non se faire. Par contre, la classe `MonPoint` est une classe où seul le tracé compte. Nous avons donc besoin d'enregistrer la couleur du tracé ainsi que le fait de savoir si l'objet doit être affiché ou non.

Nous pouvons alors écrire les définitions de ces trois classes, à savoir

```
public class MonPoint
{
    private int _X = 0, _Y = 0;
    private bool _Visible = true;
    private PictureBox _Hebergeur;
    private Color _Fond = Color.Silver, _Crayon = Color.Black;
}

public class MonRectangle : MonPoint
{
    private Color _Pot = Color.Red;
    private bool _Remplir = true;
    private int _Longueur = 1, _Hauteur = 1;
}

public class MonCercle : MonPoint
{
    private Color _Pot = Color.Red;
    private bool _Remplir = true;
    private int _Rayon = 1;
}
```


Nous noterons que la classe `MonPoint` comporte un champ de type `PictureBox`. Cet élément, commun à toutes les classes, n'est rien d'autre qu'une référence au composant dans lequel nous allons dessiner.

10.1.6. Une petite coquetterie : les accesseurs

Comme nous pouvons le constater, les données membres de classe sont précédées du symbole `_`. Cette convention prépare en fait l'utilisation d'accesseurs pour chaque données des classes ainsi définies. Pour rappel, cela nous permet de placer des *sécurités* sur les valeurs transmises.

Par exemple, nous pouvons écrire

```
public int X
{
    get
    { return this._X; }
    set
    {
        if (value < 0) this._X = 0;
        else if (value > this._Hebergeur.Bounds.Size.Width)
            this._X = this._Hebergeur.Bounds.Size.Width;
        else
            this._X = value;
    }
}
```

Cet accesseur oblige le point à se positionner à l'intérieur du composant `PictureBox`.

Les autres accesseurs sont similaires quoique souvent plus simples.

10.1.7. Description des classes : les méthodes

10.1.7.1. Introduction

Nous devons commencer par réfléchir aux actions que nous devons associer au carrosse et en déduire les méthodes à mettre en place, aussi bien pour le carrosse que pour les autres classes.

Nous devons prévoir

- la création d'un carrosse,
- l'affichage du carrosse,
- la *disparition*, au sein du `PictureBox`, du carrosse,
- le déplacement du carrosse.

L'impression de déplacement est effectué selon le principe des dessins animés. Nous utilisons une succession d'images qui ne sont rien d'autre que le carrosse qui change de place. Pour ce faire, nous commençons par afficher ce dernier, attendons 40 millièmes de seconde (25 images par seconde) puis effaçons le carrosse.

Cette succession d'actions est prise en charge par un chronomètre qui se déclenche tous les 40/1000 seconde.

Pour ce faire, nous déposons sur la fenêtre un composant de type `Timer` que nous baptisons `Images` (fenêtre `Properties`, champ `Name`) et pour lequel le champ `Interval` prend la valeur 40 (fenêtre `Properties`, champ `Interval`).

Il ne nous reste plus qu'à définir la méthode qui se déclenche une fois le laps de temps terminé. Un double clic sur le champ Tick de la fenêtre Properties (onglet Event) nous permet d'écrire

```
private void Images_Tick(object sender, System.EventArgs e)
{
    if (this.ca.X + this.ca.Longueur >= this.TV.Width)
        this.Images.Stop();
    else
    {
        this.ca.Cacher(this.TV.Handle);
        this.ca.Bouger(4, 0);
        this.ca.Afficher(this.TV.Handle);
    }
}
```

La condition qui entraîne l'arrêt du composant Images est relative à la position du carrosse dans le PictureBox et ne permet pas que le carrosse *sorte de l'écran*.

Pour démarrer le carrosse, tout se fait par l'intermédiaire d'un bouton. L'événement Click de ce dernier peut se mettre sous la forme

```
private void BtnCarrosse_Click(object sender, System.EventArgs e)
{
    this.ca = new Carrosse(this.TV, 10, 75, 100, 60);
    this.ca.Afficher(this.TV.Handle);
    this.Images.Start();
}
```

Cette procédure permet de créer le carrosse en passant par le constructeur sur lequel nous allons de suite revenir. Une fois les éléments qui le composent disposés, le programme l'affiche dans le PictureBox puis active le chronomètre par l'intermédiaire du composant Images.

Il nous reste donc à examiner quelle est la forme des méthodes Afficher(), Cacher() et Avancer() de la classe Carrosse, sans oublier le constructeur.

Les trois premières méthodes sont très simples à rédiger puisque, selon la *philosophie* orientée objet, elles vont utiliser ce qui a été défini pour chacun des éléments qui la constituent, à savoir les méthodes correspondantes des classes MonRectangle et MonCercle.

10.1.7.2. Les constructeurs

Le constructeur du carrosse est essentiellement caractérisé par le positionnement de chaque élément du carrosse par rapport au coin supérieur gauche de l'habitacle. Ainsi, par exemple, les centres des deux roues sont positionnés sur les coins inférieurs de l'habitacle et possèdent un rayon valant la moitié de la hauteur de ce dernier.

```
public Carrosse(PictureBox hebergeur, int xsg, int ysg, int lg, int ht)
: base(hebergeur, xsg, ysg, lg, ht)
{
    this.RoueG = new MonCercle(hebergeur, xsg, ysg+ht, ht/2, Color.Brown, Color.Brown);
    this.RoueD = new MonCercle(hebergeur, xsg+lg, ysg+ht, ht/2, Color.Brown, Color.Brown);
    this.FenG = new MonRectangle(hebergeur, lg/10+xsg, ht/6+ysg, lg/5, ht/3);
    this.FenD = new MonRectangle(hebergeur, xsg+lg-3*lg/10, ht/6+ysg, lg/5, ht/3);
    this.Porte = new MonRectangle(hebergeur, xsg+lg/2-2*lg/15, ysg+ht-3*ht/4-1, 4*lg/15, 3*ht/4);
    this.Poignee = new MonRectangle(hebergeur, xsg+lg/2+lg/30, ht/2+ysg, lg/15, ht/15);
    this.Crayon = Color.Red;
    this.Pot = Color.Gold;
    this.FenG.Pot = this.FenD.Pot = Color.Navy;
    this.Porte.Pot = Color.Red;
    this.Poignee.Pot = this.Poignee.Crayon = Color.Yellow;
}
```

Ceci nous ramène à définir les constructeurs adéquats pour les classes `MonRectangle`, `MonCercle` et `MonPoint`.

En ce qui concerne la classe `MonRectangle`, nous devons prévoir un constructeur permettant d'enregistrer le `PictureBox` où sera dessiné le rectangle ainsi que sa position et sa dimension. Le placement du rectangle se fait par l'intermédiaire de coordonnées (x,y) prises en charge par la classe `MonPoint`. C'est pourquoi nous faisons appel au constructeur de cette classe pour enregistrer les coordonnées du coin supérieur gauche du rectangle. Nous retenons le constructeur

```
public MonRectangle(PictureBoxhebergeur, int xsg, int ysg, int longueur, int hauteur)
: base(hebergeur, xsg, ysg)
{Longueur = longueur;
Hauteur = hauteur;
}
```

Nous noterons que ce dernier fait appel, par l'intermédiaire du mot `base()`, au constructeur de la classe `MonPoint` pour enregistrer dans l'objet de la classe `MonRectangle` les coordonnées x et y ainsi que le composant `PictureBox`. Pour rappel, puisque la classe `MonRectangle` hérite de `MonPoint`, les données de `MonPoint` sont automatiquement intégrés à chaque objet de la classe `MonRectangle`.

Pour ce qui est de la classe `MonCercle`, nous avons besoin du constructeur

```
public MonCercle(PictureBoxhebergeur, int xc, int yc, int rayon, Color crayon, Color pot)
: base(hebergeur, xc, yc, crayon)
{Rayon = rayon;
Pot = pot;
}
```

Si nous récapitulons, il est alors nécessaire de créer deux constructeurs pour la classe `MonPoint`. ils se distinguent par les arguments transmis, à savoir

- le `PictureBox` et les coordonnées x et y,
- le `PictureBox`, les coordonnées x et y ainsi que la couleur de trait.

Nous pouvons alors écrire

```
public MonPoint(PictureBoxhebergeur)
{this._Hebergeur =hebergeur;
Fond =hebergeur.BackColor;
}
public MonPoint(PictureBoxhebergeur, int x, int y)
: this(hebergeur)
{X = x;
Y = y;
}
public MonPoint(PictureBoxhebergeur, int x, int y, Color crayon)
: this(hebergeur, x, y)
{Crayon = crayon; }
```

Nous avons ajouté un constructeur ne recevant que le `PictureBox` en argument.

Nous pouvons constater que nous utilisons, pour les deux derniers constructeurs, un appel à un constructeur précédemment défini de la classe `MonPoint`, par l'intermédiaire du mot `this()`.

10.1.7.3. L'affichage

L'affichage se fait en dessinant à l'écran, dans le PictureBox, chacun des composants. Nous devons juste prendre soin de terminer par les éléments qui sont à l'avant plan. Nous avons

```
public new void Afficher(IntPtr handle)
{
    base.Afficher(handle);
    this.RoueG.Afficher(handle);
    this.RoueD.Afficher(handle);
    this.FenG.Afficher(handle);
    this.FenD.Afficher(handle);
    this.Porte.Afficher(handle);
    this.Poignee.Afficher(handle);
}
```

Signalons tout d'abord que l'instruction

```
base.Afficher(handle);
```

permet de dessiner l'habitacle du carrosse, *base* de notre classe.

Comme nous pouvons le voir, cette méthode fait appel à deux méthodes qui permettent, respectivement, d'afficher un cercle et un rectangle. La méthode retenue est la deuxième présentée en 10.1.4. Devant tenir compte des valeurs de `_Visible` (objet devant être dessiné ou non) et de `_Remplir` (objet devant être colorié ou non), ces deux dernières se mettent sous la forme

```
public void Afficher(IntPtr handle)
{
    if (this.Visible)
    {
        Graphics gr = Graphics.FromHwnd(handle);
        if (this.Remplir)
            gr.FillEllipse(new SolidBrush(this.Pot), this.X - this.Rayon, this.Y - this.Rayon,
                           2 * this.Rayon, 2 * this.Rayon);
        gr.DrawEllipse(new Pen(this.Crayon), this.X - this.Rayon, this.Y - this.Rayon,
                       2 * this.Rayon, 2 * this.Rayon);
    }
}

public void Afficher(IntPtr handle)
{
    if (this.Visible)
    {
        Graphics gr = Graphics.FromHwnd(handle);
        if (this.Remplir)
            gr.FillRectangle(new SolidBrush(this.Pot), this.X, this.Y,
                             this.Longueur, this.Hauteur);
        gr.DrawRectangle(new Pen(this.Crayon), this.X, this.Y, this.Longueur, this.Hauteur);
    }
}
```

10.1.7.4. L'effacement

Nous n'allons pas nous étendre sur cette partie puisqu'elle est similaire à ce qui précède. La seule différence concerne la couleur de crayon et de remplissage car il s'agit de la seule couleur de fond. Nous avons, en ce qui concerne le carrosse,

```
public new void Cacher(IntPtr handle)
{
    base.Cacher(handle);
    this.RoueG.Cacher(handle);
    this.RoueD.Cacher(handle);
    this.FenG.Cacher(handle);
    this.FenD.Cacher(handle);
    this.Porte.Cacher(handle);
    this.Poignee.Cacher(handle);
}
```

Notons toutefois que cette méthode est un peu trop complète puisque cacher l'habitable permet par la même occasion de cacher les fenêtres, la porte et la poignée. Nous pourrions donc nous passer des quatre dernières instructions.

Les méthodes concernant les cercles et les rectangles se mettent respectivement sous la forme

```

public void Cacher(IntPtr handle)
{Graphics gr = Graphics.FromHwnd(handle);
 gr.FillRectangle(new SolidBrush(this.Fond), this.X, this.Y, this.Longueur, this.Hauteur);
}
public void Cacher(IntPtr handle)
{Graphics gr = Graphics.FromHwnd(handle);
 gr.FillEllipse(new SolidBrush(this.Fond), this.X - this.Rayon,
               this.Y - this.Rayon, 2 * this.Rayon, 2 * this.Rayon);
 gr.DrawEllipse(new Pen(this.Fond), this.X - this.Rayon, this.Y - this.Rayon,
               2 * this.Rayon, 2 * this.Rayon);
}

```

10.1.7.5. Le déplacement

Il ne reste plus que le mouvement à proprement parler. Chaque élément du carrosse doit accepter une modifications de ses coordonnées x et y. Cependant, ces coordonnées sont enregistrées dans la classe MonPoint. C'est pourquoi nous *sautons une étape* et définissons une méthode Bouger() dans cette dernière classe.

```

public void Bouger(int deplX, int deplY)
{X += deplX;
 Y += deplY;
}

```

Pour rappel, l'appel de Bouger() associé à la classe Carrosse entraîne un appel aux méthodes Bouger() des classes MonCercle et MonRectangle. Comme ces dernières ne sont pas définies dans ces classes, le compilateur renseigne alors la méthode Bouger() de la classe dont héritent MonCercle et MonRectangle, à savoir la méthode Bouger() de MonPoint.

10.2. Gestion des nombres rationnels

Nous allons consacrer cette partie à la définition et à la mise en place de méthodes permettant de manipuler des nombres rationnels, c'est-à-dire des fractions.

10.2.1. Définition

La classe Rationnel nécessite deux données membres entières, à savoir le numérateur et le dénominateur. Nous pouvons écrire

```

public class Fraction
{private int nume;
 private int deno;
}

```

10.2.2. Les constructeurs

Une fois les données déterminées, nous pouvons passer aux constructeurs. Nous avons retenu les possibilités suivantes :

- le renseignement des numérateur et dénominateur en veillant à générer une exception, par l'instruction `throw`, lorsque l'utilisateur transmet la valeur 0 comme dénominateur,

```
public Fraction(int nume,int deno)
{if (deno == 0)
    throw new ArgumentOutOfRangeException("Le dénominateur ne peut être nul !");
else
    {this.nume = nume;
      this.deno = deno;
    }
}
```

- le renseignement d'une seule valeur entière qui est alors considérée comme étant le numérateur, le dénominateur étant quant à lui initialisé à 1,

```
public Fraction(int nume)
{this.nume = nume;
  this.deno = 1;
}
```

- le renseignement d'une fraction,

```
public Fraction(Fraction f)
{this.nume = f.nume;
  this.deno = f.deno;
}
```

- la prise en charge de la copie exige que la classe hérite de l'interface `ICloneable` ce qui engendre la redéfinition

```
public class Fraction : ICloneable
{ // Données membres et méthodes de la classe
}
```

- la copie en elle-même étant prise en charge par la méthode

```
public object Clone()
{return new Fraction(this.nume,this.deno);
}
```

10.2.3. Opérateurs arithmétiques entre fractions

Nous devons maintenant mettre en place les opérateurs arithmétiques traditionnels pour rendre cette définition utilisable, à savoir

- le changement de signe renseigné par l'opérateur `-` et portant sur un seul rationnel

```
public static Fraction operator- (Fraction f)
{return new Fraction(-f.nume,f.deno);
}
```

- l'addition `+` entre deux rationnels est évidemment basée sur la réduction au même dénominateur de ces deux rationnels

```
public static Fraction operator+ (Fraction f1,Fraction f2)
{Fraction res = new Fraction(0);
  if (f1.deno != f2.deno)
  {res.nume = f1.nume*f2.deno + f2.nume*f1.deno;
    res.deno = f1.deno*f2.deno;
  }
  else
  {res.nume = f1.nume + f2.nume;
    res.deno = f1.deno;
  }
  return res;
}
```

- la soustraction – se base sur le même calcul

```

public static Fraction operator- (Fraction f1, Fraction f2)
{
    Fraction res = new Fraction(0);
    if (f1.deno != f2.deno)
    {
        res.num = f1.num*f2.deno - f2.num*f1.deno;
        res.deno = f1.deno*f2.deno;
    }
    else
    {
        res.num = f1.num - f2.num;
        res.deno = f1.deno;
    }
    return res;
}

```

- la multiplication * s'effectue en multipliant numérateurs et dénominateurs

```

public static Fraction operator* (Fraction f1, Fraction f2)
{
    return new Fraction(f1.num*f2.num, f1.deno*f2.deno);
}

```

- la division / s'effectue en multipliant (opération venant d'être définie) la première fraction par l'inverse de la seconde tout en prenant quelques précautions pour que la division puisse s'effectuer (ne pas diviser par 0)

```

public static Fraction operator/ (Fraction f1, Fraction f2)
{
    if (f2.num == 0)
        throw new ArgumentNullException("Le diviseur ne peut être nul !");
    else
        return f1 * new Fraction(f2.deno, f2.num);
}

```

10.2.4. Opérateurs de conversion

Nous pouvons maintenant passer aux opérateurs de conversion (casting). Nous avons

- l'opérateur de conversion d'une fraction en entier s'écrit

```

public static explicit operator int (Fraction f)
{
    return f.num / f.deno;
}

```

- l'opérateur de conversion d'une fraction en réel de type float s'écrit

```

public static explicit operator float (Fraction f)
{
    return (float) f.num / f.deno;
}

```

- l'opérateur de conversion d'un entier en fraction s'écrit

```

public static implicit operator Fraction(int i)
{
    return new Fraction(i, 1);
}

```

10.2.5. Opérations arithmétiques mixtes (fraction et entier)

Nous persévérons dans la transformation de types en définissant les opérateurs arithmétiques mélangeant les entiers et les fractions. Nous commençons par l'opérateur + d'addition entre un entier et une fraction. L'addition étant commutative, nous devons définir

- l'addition du type *entier ajouté à une fraction*,
- l'addition du type *fraction ajoutée à un entier*.

En nous basant sur les opérateurs de conversion, nous pouvons écrire

```

public static Fraction operator+ (int i, Fraction f)
{
    return f + (Fraction)i;
}
public static Fraction operator+ (Fraction f, int i)
{
    return f + (Fraction)i;
}

```

Nous pouvons même définir cette addition de différente manière. Par exemple,

```
public static Fraction operator+ (int i, Fraction f)
{
    return f + new Fraction(i);
}
```

ou encore

```
public static Fraction operator+ (int i, Fraction f)
{
    return new Fraction(f.num + i*f.deno, f.deno);
}
```

Les autres opérations arithmétiques mixtes sont similaires, à savoir

- la soustraction -

```
public static Fraction operator- (int i, Fraction f)
{
    return ((Fraction)i) - f;
}
public static Fraction operator- (Fraction f, int i)
{
    return f - (Fraction)i;
}
```

- la multiplication *

```
public static Fraction operator* (int i, Fraction f)
{
    return ((Fraction)i) * f;
}
public static Fraction operator* (Fraction f, int i)
{
    return f * (Fraction)i;
}
```

- la division /

```
public static Fraction operator/ (int i, Fraction f)
{
    return ((Fraction)i) / f;
}
public static Fraction operator/ (Fraction f, int i)
{
    if(i == 0) throw new ArgumentException("Le diviseur ne peut être nul !");
    else return f / (Fraction)i;
}
```

- l'opérateur % prend la forme suivante, dans le monde rationnel,

```
public static Fraction operator% (Fraction f, int i)
{
    int tmp = (int)f;
    return (f-tmp) + tmp%i;
}
```

- l'opérateur ++ se base sur l'addition d'un entier et d'une fraction,

```
public static Fraction operator++ (Fraction f)
{
    return f + 1;
}
```

- l'opérateur -- prend la forme suivante, dans le monde rationnel,

```
public static Fraction operator-- (Fraction f)
{
    return f - 1;
}
```

10.2.6. Surcharge de la classe Object

Avant de passer à la dernière catégorie d'opérateurs, à savoir les opérateurs de comparaison, nous allons commencer par surcharger deux méthodes de la classe de base Object :

- la méthode ToString() permet de personnaliser l'affichage des fractions que nous venons de définir et peut s'écrire

```
public override string ToString()
{
    if(this.deno == 1) return String.Format("{0}", this.num);
    else return String.Format("{0}/{1}", this.num, this.deno);
}
```

- la méthode d'égalité Equals() s'écrit, quant à elle,

```
public override bool Equals(Object f)
{
    return (this == (Fraction)f);
}
```


10.2.7. Opérateurs de comparaison

En ce qui concerne les opérateurs de comparaison, nous avons classiquement

- l'opérateur ==

```
public static bool operator== (Fraction f1, Fraction f2)
{
    return (f1.num * f2.den) == (f2.num * f1.den);
}
```

- l'opérateur !=

```
public static bool operator!= (Fraction f1, Fraction f2)
{
    return !(f1 == f2);
}
```

- l'opérateur <

```
public static bool operator< (Fraction f1, Fraction f2)
{
    return (f1.num * f2.den) < (f2.num * f1.den);
}
```

- l'opérateur >

```
public static bool operator> (Fraction f1, Fraction f2)
{
    return !((f1 < f2) || (f1 == f2));
}
```

- l'opérateur <=

```
public static bool operator<= (Fraction f1, Fraction f2)
{
    return !(f1 > f2);
}
```

- l'opérateur >=

```
public static bool operator>= (Fraction f1, Fraction f2)
{
    return !(f1 < f2);
}
```

10.2.8. Prise en compte de la simplification

Nous allons maintenant ajouter une cerise sur le gâteau. Nous allons incorporer la simplification des fractions.

Les constructeurs en tiennent compte et, par conséquent, l'instruction

```
Fraction fr = new Fraction(15, 35);
```

devient équivalente à

```
Fraction fr = new Fraction(3, 7);
```

Les opérateurs arithmétiques vont utiliser la méthode `Simplifie()` définie ci-dessous pour transmettre une réponse simplifiée.

Avant toute chose, nous définissons une méthode permettant de déterminer le plus grand commun diviseur de deux nombres entiers. Cette méthode peut s'écrire

```
private static int pgcd(int n1, int n2)
{
    if(n1 < 0) n1 = -n1;
    if(n2 < 0) n2 = -n2;
    if(n1 > n2) return pgcd(n2, n1);
    else if(n1 == 0) return n2;
    else return pgcd(n2%n1, n1);
}
```

A partir de cette méthode statique, nous pouvons définir la méthode `Simplifie()` présentée auparavant selon les instructions

```
private static Fraction Simplifie(Fraction f)
{
    if (f.num == 0)
        return new Fraction(0);
    else
    {
        int div;
        div = pgcd(f.num, f.den);
        return new Fraction(f.num / div, f.den / div);
    }
}
```

Les modifications apportées aux constructeurs ne concernent en fait que le constructeur à deux arguments entiers qui devient

```
public Fraction(int num, int den)
{
    if (den == 0)
        throw new ArgumentOutOfRangeException("Le dénominateur ne peut être nul !");
    else
    {
        int div = pgcd(num, den);
        this.num = num / div;
        this.den = den / div;
    }
}
```

En ce qui concerne les opérateurs arithmétiques entre deux fractions présentés en 10.2.3, il suffit d'affecter de la méthode `Simplifie()` le résultat renvoyé. Par exemple, la définition de l'opérateur `*` prend la forme

```
public static Fraction operator* (Fraction f1, Fraction f2)
{
    return new Fraction(f1.num*f2.num, f1.den*f2.den);
}
```

Il reste les opérateurs arithmétiques mixtes présentés en 10.2.5. Le principe est identique à celui des autres opérateurs arithmétiques mais ne concerne que les opérateurs de multiplication et de division. Par exemple, nous pouvons écrire

```
public static Fraction operator* (int i, Fraction f)
{
    return Simplifie(((Fraction)i) * f);
}
```

10.3. Sérialiser et désérialiser des données par des fichiers XML

La sérialisation (Marshaling) est un concept répandu en programmation orientée objet qui consiste à encoder l'état d'un objet présent en mémoire sous la forme d'un flux de données. Ce flux de données peut prendre plusieurs formes mais il s'agit, ici, de nous pencher sur une sauvegarde dans un fichier XML, à savoir des fichiers texte balisés qui servent à transférer universellement des données entre diverses applications et même entre différents systèmes d'exploitation.

Evidemment, il faut mettre en forme les données à envoyer et se préparer à les recevoir :

- la sérialisation est la mise en format XML de données,
- la désérialisation est la récupération de données sauvegardées en format XML.

Avant de voir comment préparer, sous la forme de flux de données, les données devant subir ces opérations, nous allons commencer par voir brièvement à quoi ressemble un fichier XML. Nous présenterons ensuite les aménagements à mettre en place pour sérialiser/désérialiser des données et terminerons par les opérations d'écriture et de lecture à proprement parler.

10.3.1. Structure d'un fichier de données XML

XML (eXtensible Markup Language) est un langage de balisage comme l'est HTML. Le but de ce langage est de permettre de créer des documents structurés à l'aide de balises créées par le concepteur du document mais néanmoins universellement utilisables.

Un tel document est divisé en trois parties.

- le prologue renseigne la version utilisée de XML ainsi que l'éventuel jeu de caractères qui, dans ce qui suit, permet de prendre en compte les accents de la langue française,

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

- la déclaration facultative d'un document, par exemple Gens.dtd, de définitions de types repris dans le fichier

```
<!DOCTYPE Gens SYSTEM "Gens.dtd">
```

- les données

```
<PersonneSerialisee ID="1">
  <Nom>Winch</Nom>
  <Pre>Largo</Pre>
  <Liste>
    <Conquête>Danitza</Conquête>
    <Conquête>Charity</Conquête>
    <Conquête>Marilyn</Conquête>
  </Liste>
</PersonneSerialisee>
```

10.3.2. Préparer les données

Nous commençons par une définition *standard* de classe comme suit, les champs étant rendus accessibles en lecture et en écriture par des accesseurs publics.

```
public class PersonneSerialisee
{
    private int _ID;
    private string _Nom, _Pre;
    private DateTime _Nai;
    private List<string> _Lst;
    public PersonneSerialisee()
    {
        _Lst = new List<string>();
    }
    public PersonneSerialisee(string Nom_, string Pre_, DateTime Nai_)
    : this()
    {
        _Nom = Nom_; _Pre = Pre_; _Nai = Nai_;
    }
    public PersonneSerialisee(int ID_, string Nom_, string Pre_, DateTime Nai_)
    : this(Nom_, Pre_, Nai_)
    {
        _ID = ID_;
    }
    public int ID
    {
        get { return _ID; }
        set { _ID = value; }
    }
    public string Nom
    {
        get { return _Nom; }
        set { _Nom = value; }
    }
    public string Pre
    {
        get { return _Pre; }
        set { _Pre = value; }
    }
    public DateTime Nai
    {
        get { return _Nai; }
        set { _Nai = value; }
    }
    public List<string> Lst
    {
        get { return _Lst; }
        set { _Lst = value; }
    }
}
```

10.3.3. Les instructions pour sérialiser et désérialiser les données

Une fois cette classe `PersonneSerialisee` définie, nous devons ajouter des attributs de sérialisation pour que cette dernière puisse être effective.

Nous commençons par renseigner la classe comme pouvant être sérialisée et, dans la foulée, la définissons comme racine grâce à `XmlRoot` qui, via une chaîne de caractères en argument, peut modifier le nom de l'élément sérialisé. Nous écrivons

```
[Serializable]
[System.Xml.Serialization.XmlRoot()]
public class PersonneSerialisee
{
    // Contenu de la classe ... Rien ne change
}
```

Après avoir habillé la classe, nous pouvons maintenant préparer ses membres. Le principe est identique. Il suffit de faire précéder les accesseurs que nous souhaitons inclure dans le flux de données par les attributs adéquats. Il faut noter que les attributs publics disponibles en lecture et écriture seront alors sérialisés par défaut. Nous avons à disposition

- `XmlElement` officialise l'inclusion de l'accesseur dans la liste des éléments de la classe à sérialiser, le string transmis en argument pouvant rebaptiser l'élément,
- `XmlIgnore` empêche la sérialisation de l'élément associé comme la propriété `Nai`,
- `XmlAttribute` transforme la propriété, par exemple `ID`, en un attribut de l'élément qualifié de racine,
- `XmlArray` permet de personnaliser la liste prise en charge par la classe tandis que `XmlArrayItem` s'occupe des éléments de la liste.

Nous pouvons compléter le code relatif aux accesseurs comme suit

```
[System.Xml.Serialization.XmlAttribute("Identifiant")]
public int ID
{
    // Contenu du champ _ID
}
[System.Xml.Serialization.XmlElement("Nom")]
public string Nom
{
    // Contenu du champ _Nom
}
[System.Xml.Serialization.XmlElement("Prénom")]
public string Pre
{
    // Contenu du champ _Pre
}
[System.Xml.Serialization.XmlIgnore()]
public DateTime Nai
{
    // Contenu du champ _Nai
}
[System.Xml.Serialization.XmlArray("Liste")]
[System.Xml.Serialization.XmlArrayItem("Conquête")]
public List<string> Lst
{
    // Contenu du champ _Lst
}
```

La gestion des flux se fait dans `System.IO` tandis que les opérations de sérialisation en tant que telles sont localisées dans `System.Xml.Serialization`.

Supposons que nous souhaitions gérer les opérations en mémoire. Pour ce faire, nous utilisons un objet de type `System.IO.MemoryStream`. Dans le cas où nous voulons sauvegarder dans un fichier le résultat de la sérialisation, nous pouvons nous appuyer sur un objet de type `System.IO.StreamWriter`.

Une fois défini l'objet où doit être stocké le résultat de la sérialisation, nous nous appuyons sur un objet de type `System.Xml.Serialization.XmlSerializer`.

Nous enrichissons alors la classe `PersonneSerialisee` de deux méthodes permettant de sérialiser un objet issu de la classe, les deux versions se différenciant, comme annoncé auparavant, selon la destination de cette sérialisation.

```

public string SerialiseMem()
{
    using (MemoryStream ms = new MemoryStream())
    {
        XmlSerializer ser = new XmlSerializer(this.GetType());
        ser.Serialize(ms, this);
        ms.Close();
        return Encoding.Default.GetString(ms.ToArray());
    }
}

public void SerialiseFic(string NomFic)
{
    using (StreamWriter sw = new StreamWriter(NomFic))
    {
        XmlSerializer ser = new XmlSerializer(this.GetType());
        ser.Serialize(sw, this);
        sw.Close();
    }
}

```

Nous pouvons maintenant passer à la récupération des données depuis un flux XML que nous supposons issu d'un fichier. Pour ce faire, nous définissons la méthode suivante comme étant statique pour être accessible sans instantiation. A nouveau, un objet de type `System.Xml.Serialization.XmlSerializer` permet de réaliser les opérations de transfert entre la forme XML et les classes C#.

```

public static PersonneSerialisee Deserialise(string NomFichier)
{
    StreamReader fic = new StreamReader(NomFichier);
    XmlSerializer s = new XmlSerializer(typeof(PersonneSerialisee));
    PersonneSerialisee rep = (PersonneSerialisee)s.Deserialize(fic);
    fic.Close();
    return rep;
}

```

Pour en terminer avec ces manipulations, nous n'avons plus qu'à traiter les appels à ces ressources. Le remplissage suivant de la méthode `Main()` convient parfaitement pour sérialiser un objet de type `PersonneSerialisee` préalablement défini.

```

static void Main(string[] args)
{
    PersonneSerialisee p
    = new PersonneSerialisee(1, "Winch", "Largo", new DateTime(1975, 07, 15));
    p.Lst.Add("Danitza");
    p.Lst.Add("Charity");
    p.Lst.Add("Marilyn");
    p.SerialiseFic("Essai.xml");
}

```

Le résultat obtenu ressemble à

```

<?xml version="1.0" encoding="utf-8"?>
<PersonneSerialisee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    Identifiant="1">
  <Nom>Winch</Nom>
  <Prénom>Largo</Prénom>
  <Liste>
    <Conquête>Danitza</Conquête>
    <Conquête>Charity</Conquête>
    <Conquête>Marilyn</Conquête>
  </Liste>
</PersonneSerialisee>

```

Pour récupérer les données ainsi enregistrées, il suffit de l'instruction

```

PersonneSerialisee p2 = PersonneSerialisee.Deserialise("Essai.xml");

```

10.3.4. Une sérialisation plus directe

Ce qui précède suppose un traitement de la classe pour préparer les données à être sérialisées. Il est également possible de réaliser une sérialisation semblable sans passer par ce travail préliminaire.

La classe de données `Personne` sur laquelle nous nous appuyons est une version simplifiée de la classe `PersonneSerialisee`. Nous ne retenons que les champs `_ID`, `_Nom`, `_Pre` et `_Lst` avec les constructeurs et accesseurs correspondant. La mention des attributs de sérialisation n'est évidemment pas maintenue.

Commençons par sauvegarder les données en remplaçant, par le code, les balises utilisées précédemment. Un nœud de *contenu* est renseigné par la méthode `WriteStartElement()` qui entraîne l'apparition de la balise ouvrante tandis que la méthode `WriteEndElement()` met en place la balise fermante. Nous les utilisons pour signaler l'élément principal et la liste de `string`. Nous écrivons la méthode comme suit.

```
public void SerialiseFic(string NomFic, Encoding Enc)
{
    using (System.Xml.XmlTextWriter xw = new System.Xml.XmlTextWriter(NomFic, Enc))
    {
        xw.WriteStartDocument();
        xw.WriteStartElement("Personne");
        xw.WriteAttributeString("Identifiant", ID.ToString());
        xw.WriteElementString("Nom", Nom);
        xw.WriteElementString("Prénom", Pre);
        xw.WriteStartElement("Liste");
        foreach (string e in this.Lst)
            xw.WriteElementString("Conquête", e);
        xw.WriteEndElement();
        xw.WriteEndElement();
        xw.Close();
    }
}
```

Le fichier généré est le suivant.

```
<?xml version="1.0" encoding="UTF-8"?>
<Personne Identifiant="1">
  <Nom>Winch</Nom>
  <Prénom>Largo</Prénom>
  <Liste>
    <Conquête>Danitza</Conquête>
    <Conquête>Charity</Conquête>
    <Conquête>Marilyn</Conquête>
  </Liste>
</Personne>
```

La récupération en mémoire des données à partir du fichier peut se faire comme suit.

```
public static Personne Deserialise(string NomFichier)
{
    Personne rep = new Personne();
    System.Xml.XmlTextReader xr = new System.Xml.XmlTextReader(NomFichier);
    while (xr.Read())
    {
        if (xr.Name == "Personne")
        {
            xr.MoveToAttribute("Identifiant");
            rep.ID = xr.ReadContentAsInt();
            xr.Read();
            rep.Nom = xr.ReadElementContentAsString();
            rep.Pre = xr.ReadElementContentAsString();
            if (xr.Name == "Liste" && !xr.IsEmptyElement)
            {
                xr.Read();
                while (xr.Name == "Conquête")
                    rep.Lst.Add(xr.ReadElementContentAsString());
            }
            xr.Read();
        }
    }
    xr.Close();
    return rep;
}
```

Pour utiliser ces ressources, nous pouvons écrire le code suivant dans la méthode `Main()`.

```

Personne p = new Personne(1, "Winch", "Largo");
p.Lst.Add("Danitza");
p.Lst.Add("Charity");
p.Lst.Add("Marilyn");
p.SerializeFic("EssaiSimple.xml", Encoding.UTF8);

```

La récupération, quant à elle, peut se mettre sous la forme suivante.

```

Personne p2 = Personne.Deserialize("EssaiSimple.xml");

```

10.3.5. Une sérialisation brute

Nous pouvons utiliser une méthode encore plus simple qui, en contrepartie, ne nous permet pas de configurer la sauvegarde en personnalisant les champs XML. Par conséquent, nous obtenons une image tout à fait conforme aux données de la classe créée à cet effet et à sa définition.

Comme dans le cas de la mise en place des balises rencontrée en 10.3.2, nous utilisons la classe `System.Xml.Serialization.XmlSerializer`.

Pour sauvegarder les données dans un fichier XML, nous pouvons écrire le code ci-dessous, identique à celui rencontré précédemment.

```

public void SerializeFic(string NomFic)
{
    XmlSerializer xs = new XmlSerializer(this.GetType());
    StreamWriter wr = new StreamWriter(NomFic);
    xs.Serialize(wr, this);
    wr.Close();
}

```

Le fichier obtenu est le suivant et est une pure transcription en XML de la définition et du contenu de la classe `Personne`.

```

<?xml version="1.0" encoding="utf-8"?>
<Personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ID>1</ID>
  <Nom>Winch</Nom>
  <Pre>Largo</Pre>
  <Lst>
    <string>Danitza</string>
    <string>Charity</string>
    <string>Marilyn</string>
  </Lst>
</Personne>

```

La récupération se fait également de manière identique. Pour rappel, nous pouvons écrire le code suivant.

```

public static Personne Deserialize(string NomFichier)
{
    StreamReader fic = new StreamReader(NomFichier);
    XmlSerializer s = new XmlSerializer(typeof(Personne));
    Personne rep = (Personne)s.Deserialize(fic);
    fic.Close();
    return rep;
}

```

La manipulation de ces ressources dans `Main()` peut se mettre sous la forme suivante.

```

Personne p = new Personne(1, "Blackman", "Catherine");
p.SerializeFic("EssaiSimple.xml");

```

La récupération à partir de données enregistrées est identique à celle du paragraphe précédent, à savoir

```

ps = Personne.Deserialize("EssaiSimple.xml");

```

10.3.6. Généralisation de la sérialisation brute

Evidemment, la sérialisation porte rarement sur un seul élément mais plutôt sur des *collections* d'objets.

Une solution consiste à adopter les développements précédents en passant par une classe intermédiaire qui hériterait, par exemple, de `List<PersonneSerialisee>`. Evidemment, il nous faut alors développer autant de classes que nous aurons de collections à traiter. L'utilisation d'une classe générique devient alors une piste à creuser.

Nous définissons donc une classe `UtilitaireSerialisation` qui risque d'être bien utile dans le cas où la sérialisation *brute* devient le quotidien de notre travail. Conformément à ce qui est présenté en 6.14, nous allons définir, par la sérialisation, une méthode travaillant sur un type générique `T` et recevant, en argument, un nom de fichier sous la forme d'un `string` et un objet de type `T`.

```
• {public static void SerialiseObjet<T>(string NomFic, T obj)
• { }
```

Il en est de même pour la désérialisation.

Pour ce qui est du code, nous allons récupérer celui présenté en 10.3.3 et l'adapter au contexte *générique*. Nous pouvons finalement écrire la classe sous la forme définitive suivante.

```
• public class UtilitaireSerialisation
• {public static void SerialiseObjet<T>(string NomFic, T obj)
• {StreamWriter sw = new StreamWriter(NomFic);
• XmlSerializer ser = new XmlSerializer(typeof(T));
• ser.Serialize(sw, obj);
• sw.Close();
• }
• public static T DeserialiseObjet<T>(string NomFichier)
• {StreamReader fic = new StreamReader(NomFichier);
• XmlSerializer s = new XmlSerializer(typeof(T));
• T rep = (T)s.Deserialize(fic);
• fic.Close();
• return rep;
• }
• }
```

L'utilisation de ces ressources peut se faire via un code ressemblant à ce qui suit.

```
• List<Personne> ls = new List<Personne>();
• Personne pg = new Personne(1, "Winch", "Largo");
• pg.Lst.Add("Aw1"); pg.Lst.Add("Bw1"); pg.Lst.Add("Cw1");
• ls.Add(pg);
• pg = new Personne(2, "Ovronnaz", "Simon");
• pg.Lst.Add("Aos"); pg.Lst.Add("Bos"); pg.Lst.Add("Cos");
• ls.Add(pg);
• UtilitaireSerialisation.SerialiseObjet<List<Personne>>("Liste.xml", ls);
• List<Personne> lr = UtilitaireSerialisation.DeserialiseObjet<List<Personne>>("Lst.xml");
• foreach (Personne pgd in lr)
• {Console.WriteLine(pgd.Pre + " " + pgd.Nom + " (" + pgd.ID.ToString() + ")");
• foreach (string e in pgd.Lst)
• Console.WriteLine(" - " + e);
• }
```


10.3.7. Linq et XML

Il existe une autre manière d'aborder la sérialisation des éléments et cette dernière passe par une utilisation élégante de Linq.

Nous allons nous appuyer sur des ressources définies dans l'espace de nom

`System.Xml.Linq`

Dans l'exemple que nous allons traiter, nous nous limitons à quelques fonctionnalités définies dans cet espace de nom et laissons au lecteur intéressé le soin de découvrir d'autres ressources mises à sa disposition dans ce cadre.

Nous nous appuyons sur des données définies comme en 10.3.4. Nous pouvons donc définir un élément comme précédemment via le code suivant.

```
Personne p = new Personne(1, "Winch", "Largo");  
p.Lst.Add("Danitza");  
p.Lst.Add("Charity");  
p.Lst.Add("Marilyn");
```

Nous allons commencer par écrire une méthode, dans la classe `Personne` pour créer un nœud XML constitué des différents données constituant l'élément, hors la liste des conquêtes de ce même élément. La classe sur laquelle nous nous appuyons pour ce faire est `XElement` et nous pouvons écrire le code suivant.

```
public XElement ToXMLElement()  
{XElement elt = new XElement("Personne",  
                               new XElement("ID", ID),  
                               new XElement("Nom", Nom),  
                               new XElement("Pre", Pre));  
  return elt;  
}
```

Comme nous pouvons le constater, dans l'état actuel des choses, un `XElement` est composé du nom du nœud et de la valeur à enregistrer, cette valeur pouvant consister en plusieurs autres objets de type `XElement`.

Il reste maintenant à générer, dans un nœud que nous baptiserons `Lst`, l'ensemble des conquêtes de la personne. Pour ce faire, il faut en fait générer autant de `XElement` qu'il y a de conquêtes, chacun de ces objets étant caractérisés par le nom (`Conquete`) et la valeur (à récupérer dans le champ `Lst` de la personne).

Pour ce faire, nous allons utiliser le résultat d'une simple requête Linq qui parcourt le champ `Lst` pour en récupérer les valeurs `Conquete` enregistrées.

Sans nous préoccuper du contexte XML de la situation, nous pourrions écrire cette requête sous la forme suivante, en y ajoutant la boucle de parcours permettant l'affichage de ce qui a été récupéré.

```
var b = from a in p.Lst select a;  
foreach (string c in b) Console.WriteLine(c);
```

Nous pouvons adapter cette instruction pour récupérer le nom de chaque conquête sous la forme d'un objet de type `XElement` et intégrer le résultat obtenu directement dans un objet de type `XElement` de nom `Lst`.

Nous complétons alors l'instruction précédente de création d'un objet de type `XElement` sous la forme suivante.

```
public XElement ToXMLElt()
{
    XElement elt = new XElement("Personne",
        new XElement("ID", ID),
        new XElement("Nom", Nom),
        new XElement("Pre", Pre),
        new XElement("Lst", from a in Lst select new XElement("Conquete", a)));
    return elt;
}
```

La classe `XElement` n'est pas la seule disponible dans ce contexte. Par exemple, nous pouvons faire *glisser* le champ ID comme attribut du nœud de type `Personne` créé et ajouter un commentaire sur la liste des conquêtes via, respectivement, les classes `XAttribute` et `XComment`. La méthode peut alors se mettre sous la forme suivante.

```
public XElement ToXMLElt()
{
    XElement elt = new XElement("Personne",
        new XAttribute("ID", ID),
        new XElement("Nom", Nom),
        new XElement("Pre", Pre),
        new XElement("Lst", new Comment("Liste des conquêtes"),
            from a in Lst select new XElement("Conquete", a)));
    return elt;
}
```

Maintenant que nous avons défini cette méthode, nous pouvons passer à la création d'une fonction qui encapsule cet élément dans un document XML via la classe `XDocument`. Cette dernière, qui utilise une nouvelle classe `XDeclaration`, peut se mettre sous la forme suivante.

```
public XDocument ToXMLDoc()
{
    return new XDocument(new XDeclaration("1.0", "utf-8", "yes"), ToXMLElt());
}
```

Nous pouvons placer, à la suite de l'instanciation de l'objet `p` dans la méthode `Main()`, l'instruction suivante pour voir directement à l'écran le résultat de la sérialisation.

```
Console.WriteLine(p.ToXMLDoc());
```

Pour conserver ce résultat de manière permanente, il nous suffit de transférer cet affichage dans un fichier par la méthode `Save` qui peut être directement appliquée à `p.ToXMLDoc()` de type `XDocument`. L'instruction pour ce faire est reprise ci-dessous.

```
p.ToXMLDoc().Save(@"c:\essai.xml");
```

Si nous ouvrons le fichier ainsi sauvegardé, nous visualisons le résultat suivant.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Personne ID="1">
  <Nom>Winch</Nom>
  <Pre>Largo</Pre>
  <Lst>
    <!--Liste des conquêtes-->
    <Conquete>Danitza</Conquete>
    <Conquete>Charity</Conquete>
    <Conquete>Marilyn</Conquete>
  </Lst>
</Personne>
```

Nous pouvons maintenant passer à la sérialisation de plusieurs éléments de type `Personne`. Le principe est identique à ce que nous venons de faire, envoyant dans un objet de type `XDocument` des objets de type `XElement` obtenus à partir de la méthode définie ci-dessus selon le même principe que celui utilisé dans la méthode `ToXMLElt()` pour récupérer les différents éléments de `Lst`.

Nous pouvons alors écrire, dans `Main()`, la création d'une liste d'objets comme nous l'avons fait auparavant. Supposons que cette liste soit baptisée `ls`.

Pour varier les plaisirs, nous allons afficher le résultat de la sérialisation à l'écran. Le principe est, via Linq, de créer autant de nœuds `Personne` qu'il y a d'éléments dans `ls`. Pour ce faire, nous allons utiliser, sur chaque nœud, la fonction `ToXMLelement()`. Nous écrivons donc le code suivant

```

XElement xmlfromlist = new XElement("ListePersonne",
    from a in ls
    select new XElement("Personne",
        new XElement("Name", a.ToXMLelement())));
Console.WriteLine(xmlfromlist);

```

Le résultat à l'écran est similaire à ce qui suit.



Il nous reste maintenant à voir comment récupérer tout ou partie des données stockées dans un fichier XML. Pour ce faire nous pouvons commencer par sauvegarder dans un fichier ce que nous venons d'obtenir à l'écran. Cela peut se mettre sous la forme suivante.

```

new XDocument(new XDeclaration("1.0", "utf-8", "yes"), xmlfromlist).Save(@"e:\liste.xml");

```

Pour récupérer la seule liste des noms des personnes enregistrées, nous pouvons utiliser la requête Linq suivante.

```

var noms = (from personnes in XDocument.Load(@"e:\liste.xml").Descendants("Personne")
    select new { Nom = personnes.Element("Nom").Value }).ToList();

```

Nous constatons que la source de données est le fichier `e:\liste.xml` et que, dans les nœuds qui dépendent du nœud de type `Personne`, nous allons rechercher la valeur associée au champ `Nom` que nous baptisons de manière identique. Pour plus de facilité de traitement, nous transformons le résultat obtenu en liste générique et stockons le résultat dans la variable `noms` typée implicitement. Pour visualiser le résultat de cette récupération de données, il nous suffit de l'instruction *classique* suivante.

```

foreach (var o in noms)
    Console.WriteLine(o.Nom);

```

Nous pouvons évidemment ajouter une condition à notre récupération. Nous allons nous concentrer sur Simon Ovrinnaz en demandant à récupérer l'élément du fichier dont l'identifiant stocké est 2. Pour ce faire, nous passons par une variable `nCrit` définie en tant que `int` dans notre code C#.

```
var nomsCrit = (from personnes in XDocument.Load(@"e:\liste.xml").Descendants("Personne")
let id = int.Parse(personnes.Attribute("ID").Value)
where id.Equals(nCrit)
select new { Nom = personnes.Element("Nom").Value }).ToList();
```

La différence avec le cas précédent réside actuellement dans l'utilisation d'une variable Linq baptisée `id` qui stocke la valeur de l'attribut pour pouvoir la comparer avec `nCrit`.

Dans ce cas de figure, récupérer sur le critère de la valeur d'un identifiant revient à récupérer une seule fiche. Nous pouvons donc transformer le résultat de la requête Linq via `Single()` plutôt que `ToList()`. Cela nous permet de remplacer l'instruction de parcours de collection pour visualiser le résultat par la seule instruction suivante.

```
Console.WriteLine(nomsCrit.Nom);
```

10.3.8. La gestion des fichiers XML

Nous terminons le passage en revue des possibilités offertes par XML en nous penchant sur les ressources mises à notre disposition pour manipuler les fichiers associés, à savoir pour ajouter, supprimer et modifier.

Commençons par ajouter un nouvel élément au fichier. Définissons tout d'abord le nouvel élément par l'intermédiaire du code suivant.

```
Personne Nouveau = new Personne(3, "Blackman", "Catherine")
Nouveau.Lst.Add("Kevin Costner")
Nouveau.Lst.Add("Gerard Gere")
```

Nous pouvons maintenant passer à l'ajout à proprement parler. Pour ce faire, nous devons charger le fichier dans un objet de type `XDocument`. Nous pouvons alors l'ajouter à ce même objet puis le sauvegarder par l'intermédiaire de la méthode `Save()`. Nous écrivons ce qui suit.

```
XDocument ManipFichier = XDocument.Load(@"e:\liste.xml")
ManipFichier.Element("ListePersonne").Add(Nouveau.ToXmlElement())
ManipFichier.Save(@"e:\liste.xml")
```

Quant à la suppression, il faut évidemment charger puis sauvegarder les données par l'intermédiaire d'un objet de même type `XDocument`. Les instructions sont identiques au cas de l'ajout.

Entretemps, nous devons ôter un élément parmi ceux chargés. Nous effectuons cette recherche sur `Root.Elements` plutôt que sur `root.Descendants` car, de cette manière, nous accédons aux *enfants immédiats*. Nous utilisons l'attribut `ID` pour déterminer de manière équivoque l'élément à supprimer en utilisant une expression lambda, utilisée en lieu et place du prédicat requis.

Le prédicat de filtre est donc défini sur la recherche d'un élément dont la valeur de l'attribut `ID` vaut 2 sous la forme suivante.

```
e => e.Attribute("ID").Value.Equals("2")
```

Son intégration dans la méthode `Where` se met telle quelle. Nous utilisons ensuite la conversion sur un seul élément comme déjà rencontré suivi de la méthode `Remove` pour supprimer effectivement l'élément trouvé. Nous écrivons le code suivant.

```
ManipFichier.Root.Elements().Where(e => e.Attribute("ID").Value.Equals("2"))  
    .Single().Remove();
```

Il ne reste plus que la modification à gérer. Pour ce faire, nous encadrons à nouveau les instructions *vitales* par un chargement puis une sauvegarde des données comme précédemment. L'opération de modification à proprement parler consiste à trouver l'enregistrement, à nouveau par la méthode `where`, que nous souhaitons modifier puis lui appliquer la modification via la méthode `SetValue`.

Les expressions lambdas sont à nouveau à l'ordre du jour. Celle relative à la sélection est similaire à celle vue lors de la suppression. Nous allons rechercher la personne dont le nom est `Winch` pour remplacer son prénom `Largo` en `Nerio`. Cela peut se faire via le prédicat suivant.

```
e => e.Element("Nom").Value.Equals("Winch")
```

Pour ce qui est de la selection, nous pouvons désigner le champ sur lequel nous souhaitons effectuer la modification via l'expression ci-dessous.

```
e => e.Element("Pre")
```

Nous mettons tout ensemble sous la forme suivante.

```
ManipFichier.Root.Elements().Where(e => e.Element("Nom").Value.Equals("Winch"))  
    .Select(e => e.Element("Pre")).Single()  
    .SetValue("Nerio");
```

11. Références

C# et .NET, Gérard Leblanc, Eyrolles, 2002.

C# et .NET, Version 2, Gérard Leblanc, Eyrolles, 2006.

C# Concisely, Judith Bishop, Nigel Horspool, Addison Wesley, 2004.

CA-Visual Objects for Microsoft Windows, Version 1.0, South Seas Adventures, an exploration of CA-Visual Objects, November 1994,

CA-Visual Objects for Microsoft Windows, Version 1.0, Getting started, November 1994,

Créer ses propres classes génériques, Benjamin De Vuyst, <http://bdevuyst.developpez.com>, mai 2009,

Expressions régulières, <http://www.regular-expressions.info>,

JBuilder 2, Formation en 21 jours, David Acremann, Stéphane Dupin, Gilles Moujeard, Macmillan, 1998.

La sérialisation (Marshaling) en XML en C# .Net 2.0, Emmanuel Blancquard, www.devparadise.com.

Les nouveautés du langage C# 3.0, James Ravaille, Association DotNet France.

Linq to XML, Association DotNet France.

Linq to XML, <http://www.dreamincode.net>.

Manuel de référence Microsoft Visual C# .NET, Mickey Williams, Microsoft Press, 2002.

Programmation en C#, Microsoft Official Curriculum, 2002.

Programmation Orientée Objet, <http://www.commentcamarche.net>.

Programming in the .NET Environment, Damien Watkins, Mark Hammond, Brad Abrams, Addison Wesley, 2003.

SQL Server 2000, Marc Israël, Eyrolles, 2001.

The Complete Visual C# Programmer's Guide, Bulent Ozkir, John Schofield, Levent Camlibel, Mahesh Chand, Mike Gold, Saurabh Nandu, Shivani Maheshwari, Srinivasa Sivakamur, Microgold Press, 2002.

Tutorial C#, Robert-Michel di Scala, www.developpez.com.

Utilisation des expressions régulières en .Net, Louis-Guillaume Morand, 2005.

Visual Studio .NET in 21 days, Jason Beres, Sams Publishing, 2003.

Visual Studio .NET 2003, Aide en ligne (basée sur .NET Framework version 1.1.4322), 15 novembre 2002.

12. Table des matières

0. Introduction	1
0.1. Historique	2
0.2. Contexte	3
0.3. Présentation	4
1. La programmation orientée objet (POO).....	5
1.1. Programmation structurée	5
1.2. Améliorations à apporter	5
1.3. Pourquoi le software ne suit-il pas le hardware ?	6
1.4. La programmation orientée objet	6
1.5. Les outils disponibles	9
1.6. Conséquences sur la modélisation	11
1.7. Et au niveau informatique	12
2. L'environnement .NET	15
2.1. Manipulations en dehors de Visual Studio .NET	16
2.2. Manipulation à l'aide de Visual Studio .NET	17
3. Généralités	26
3.1. Le premier programme C#	26
3.2. Premier contact avec C#	27
3.3. Les commentaires	33
4. Les types	36
4.1. Les identificateurs en C#	36
4.2. Les types de données	36
4.3. La manipulation des variables et des constantes	38
4.4. Le type énuméré	40
4.5. Les structures	41
4.6. Les tableaux	42
5. Les instructions	45
5.1. Les blocs d'instructions	45
5.2. Les initialisations et les affectations	45
5.3. Des variables <i>atypiques</i>	47
5.4. Opérations sur les variables	47
5.5. Le branchement conditionnel (simple)	49
5.6. Le branchement conditionnel multiple	50
5.7. La boucle <i>statique</i>	51
5.8. La boucle <i>dynamique</i> (condition en entrée de boucle)	52
5.9. La boucle <i>dynamique</i> (condition en sortie de boucle)	52
5.10. L'instruction de parcours de collection	53
5.11. Les instructions de saut	54

6. Les fonctions (méthodes)	58
6.1. Définition	58
6.2. Utilisation	59
6.3. Les instructions associées à la méthode	60
6.4. Mode de transmission d'arguments	60
6.5. Le passage d'arguments par valeur	61
6.6. Le passage d'arguments par référence	62
6.7. Les paramètres de sortie	62
6.8. Arguments en nombre variable	63
6.9. La récursivité	64
6.10. La surcharge de méthode	64
6.11. Les méthodes et les tableaux	65
6.12. Les délégués	66
6.13. Les méthodes anonymes	68
6.14. Les expressions lambda	68
6.15. Les méthodes génériques	69
7. Les classes	70
7.1. Introduction	70
7.2. Les méthodes d'une classe	73
7.3. L'héritage	80
7.4. La classe de base object	85
7.5. Manipulations d'objets	86
7.6. Les structures dynamiques	87
7.7. Les définitions partielles de classes	88
7.8. Les classes génériques	89
7.9. Les méthodes d'extension	93
8. Linq	94
8.1. Introduction	94
8.2. Exemple introductif	94
8.3. Linq et les collections d'objets	95
9. Les expressions régulières	99
9.1. Introduction	99
9.2. Syntaxe	99
9.3. Les expressions régulières en C#	103
10. Annexes	106
10.1. Premier contact avec la P.O.O. : Programmation graphique sous GDI+	106
10.2. Gestion des nombres rationnels	117
10.3. Sérialiser et désérialiser des données par des fichiers XML	122
11. Références	134
12. Table des matières	135