Chapitre 4
Les bases du langage

1. Introduction

Comme tous les langages de programmation, C# impose certaines règles au développeur. Ces règles se manifestent au travers de la syntaxe du langage mais elles couvrent le large spectre fonctionnel proposé par C#. Avant d'explorer en profondeur les fonctionnalités du langage au chapitre suivant, nous étudierons donc les notions essentielles et fondamentales de C# : la création de données utilisables par une application et le traitement de ces données.

2. Les variables

Les données utilisables dans un programme C# sont représentées par des variables. Une variable est un espace mémoire réservé auquel on assigne arbitrairement un nom et dont le contenu est une valeur dont le type est fixé. On peut manipuler ce contenu dans le code en utilisant le nom de la variable.

2.1 Nommage des variables

La spécification du langage C# établit quelques règles à prendre en compte lorsque l'on nomme une variable :

– Le nom d'une variable ne peut comporter que des chiffres, des caractères de l'alphabet latin accentués ou non, le caractère ç ou les caractères spéciaux _ et μ.

– Le nom d'une variable ne peut en aucun cas commencer par un chiffre. Il peut en revanche en comporter un ou plusieurs à toute autre position.

– Le langage est sensible à la casse, c'est-à-dire qu'il fait la distinction entre majuscules et minuscules : les variables `unevariable` et `uneVariable` sont donc différentes.

– La longueur d'un nom de variable est virtuellement illimitée : le compilateur ne remonte pas d'erreur lorsqu'il est composé de 30 000 caractères ! Il n'est évidemment pas recommandé d'avoir des noms de variables aussi longs, le maximum en pratique étant plus souvent de l'ordre de la trentaine de caractères.

– Le nom d'une variable ne peut pas être un mot-clé du langage. Il est toutefois possible de préfixer un mot-clé par un caractère autorisé ou par le caractère @ pour utiliser un nom de variable similaire.

Les noms de variables suivants sont acceptés par le compilateur C# :

– `maVariable`

– `maVariableNumero1`

– `@void` (`void` est un mot-clé de C#)

– `µn3_VàRïãbl3`

De manière générale, il est préférable d'utiliser des noms de variables explicites, c'est-à-dire permettant de savoir à quoi correspond la valeur stockée dans la variable, comme `nomClient`, `montantAchat` ou `ageDuCapitaine`.

2.2 Type des variables

Une des caractéristiques de C# est la notion de typage statique : chaque variable correspond à un type de données et ne peut en changer. De plus, ce type doit être déterminable au moment de la compilation.

2.2.1 Types valeurs et types références

Les différents types utilisables en C# peuvent être décomposés en deux familles : les types valeurs et les types références. Cette notion peut être déconcertante au premier abord, puisqu'une variable représente justement une donnée et donc une valeur. Ce concept est en fait lié à la manière dont est stockée l'information en mémoire.

Lorsque l'on utilise une variable de type valeur, on accède directement à la zone mémoire stockant la donnée. Au moment de la création d'une variable de type valeur, une zone mémoire de la taille correspondant au type est allouée. Chaque octet de cette zone est automatiquement initialisé avec la valeur binaire 00000000. Notre variable aura donc une suite de 0 pour valeur binaire.

Dans le cas d'une variable de type référence, le comportement est différent. La zone mémoire allouée à notre variable contient une adresse mémoire à laquelle est stockée la donnée. On passe donc par un intermédiaire pour accéder à notre donnée. L'adresse mémoire est initialisée avec la valeur spéciale `null`, qui ne pointe sur rien, tandis que la zone mémoire contenant les données n'est pas initialisée. Elle sera initialisée lorsque la variable sera instanciée. Dans le même temps, l'adresse mémoire stockée dans notre variable sera mise à jour.

Une variable de type référence pourra donc ne contenir aucune donnée, tandis qu'une variable de type valeur aura forcément une valeur correspondant à une suite de 0 binaires.

Cette différence de fonctionnement a une conséquence importante : la copie de variable se fait par valeur ou par référence, ce qui signifie qu'une variable de type valeur sera effectivement copiée, tandis que pour un type référence, c'est l'adresse que contient la variable qui sera copiée, et il sera donc possible d'agir sur la donnée réelle indifféremment à partir de chacune des variables pointant sur ladite donnée.

2.2.2 Types intégrés

La plateforme .NET embarque plusieurs milliers de types différents utilisables par les développeurs. Parmi ces types, nous en avons une quinzaine que l'on peut considérer comme fondamentaux : ce sont les types intégrés (aussi nommés types primitifs). Ce sont les types de base à partir desquels sont construits les autres types de la BCL ainsi que ceux que le développeur crée dans son propre code. Ils permettent de définir des variables contenant des données très simples.

Ces types ont comme particularité d'avoir chacun un alias intégré à C#.

Types numériques

Ces types permettent de définir des variables numériques entières ou décimales. Ils couvrent des plages de valeurs différentes et ont chacun une précision spécifique. Certains types seront donc plus adaptés pour les calculs entiers, d'autres pour les calculs dans lesquels la précision décimale est très importante, comme les calculs financiers.

Les différents types numériques sont énumérés ci-dessous avec leurs alias ainsi que les plages de valeurs qu'ils couvrent.

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Byte`	`byte`	0 à 255	1 octet
`System.SByte`	`sbyte`	-128 à 127	1 octet
`System.Int16`	`short`	-32768 à 32767	2 octets
`System.UInt16`	`ushort`	0 à 65535	2 octets
`System.Int32`	`int`	-2147483648 à 2147483647	4 octets
`System.UInt32`	`uint`	0 à 4294967295	4 octets
`System.Int64`	`long`	-9223372036854775808 à 9223372036854775807	8 octets
`System.UInt64`	`ulong`	0 à 18446744073709551615	8 octets
`System.Single`	`float`	±1,5e-45 à ±3,4e38	4 octets

Type	Alias C#	Plage de valeurs couverte	Taille en mémoire
`System.Double`	`double`	±5,0e-324 à ±1,7e308	8 octets
`System.Decimal`	`decimal`	±1,0e-28 à ±7,9e28	16 octets

Les types numériques primitifs sont tous des types valeurs. Une variable de type numérique et non initialisée par le développeur aura pour valeur par défaut 0.

Remarque

.NET 4 a apporté le type `System.Numerics.BigInteger` afin de manipuler des entiers d'une taille arbitraire. Ce type est aussi un type valeur, mais il ne fait pas partie des types intégrés.

Les valeurs numériques utilisées au sein du code peuvent être définies à l'aide de trois bases numériques :

– **Décimale** (base 10) : c'est la base numérique utilisée par défaut.

  Exemple : 280514

– **Héxadécimale** (base 16) : elle est très utilisée dans le monde du Web, pour la définition de couleurs, et plus généralement pour encoder des valeurs numériques dans un format plus condensé. **Une valeur héxadécimale est écrite en la préfixant par 0x.**

  Exemple : 0x0447C2

– **Binaire** (base 2) : cette base est celle qui permet de s'approcher au plus près de la manière dont la machine interprète le code compilé. L'utilisation du binaire en C# peut par conséquent se révéler précieuse dans le cadre d'optimisations (décalages de bits au lieu de multiplications par 2, stockage de multiples données au sein d'une même variable, etc.). **Les valeurs binaires sont préfixées par 0b.**

  Exemple : 0b01000100011111000010

La lecture de valeurs numériques peut s'avérer difficile dans certains cas, notamment lorsqu'elles sont représentées sous forme binaire. L'équipe en charge des évolutions de C# a intégré, dans la septième version du langage, la possibilité d'ajouter des séparateurs dans les valeurs numériques afin d'améliorer la lisibilité globale. Le caractère utilisé pour cela est _ :

– 280_514
– 0x04_47_C2
– 0b0100_0100_0111_1100_0010

Types textuels

Il existe dans la BCL deux types permettant de manipuler des caractères Unicode et des chaînes de caractères Unicode : `System.Char` et `System.String`. Ces types ont respectivement pour alias `char` et `string`.

Le type `char` est un type valeur encapsulant les mécanismes nécessaires au traitement d'un caractère Unicode. Par conséquent, une variable de type char est stockée en mémoire sur deux octets.

Les valeurs de type `char` doivent être encadrées par les caractères ' ' : 'a'.

Certains caractères ayant une signification particulière pour le langage doivent être utilisés avec le caractère d'échappement \ afin d'être correctement interprétés. D'autres caractères n'ayant pas de représentation graphique doivent être aussi déclarés avec des séquences spécifiques. Le tableau suivant résume les séquences qui peuvent être utilisées.

Séquence d'échappement	Caractère associé
\'	Simple quote '
\"	Double quote "
\\	Backslash \
\a	Alerte sonore
\b	Retour arrière
\f	Saut de page
\n	Saut de ligne

Chapitre 2
La conception orientée objet

1. Approche procédurale et décomposition fonctionnelle

Avant d'énoncer les bases de la programmation objet nous allons revoir l'approche procédurale à l'aide d'un exemple concret d'organisation de code.

La programmation procédurale est un paradigme de programmation considérant les différents acteurs d'un système comme des objets pratiquement passifs qu'une procédure centrale utilisera pour une fonction donnée.

Prenons l'exemple de la distribution d'eau courante dans nos habitations et essayons d'en émuler le principe dans une application très simple. L'analyse procédurale (tout comme l'analyse objet d'ailleurs) met en évidence une liste d'objets qui sont :

– le robinet de l'évier ;
– le réservoir du château d'eau ;
– un capteur de niveau d'eau avec contacteur dans le réservoir ;
– la pompe d'alimentation puisant l'eau dans la rivière.

Le code du programme "procédural" consisterait à créer un ensemble de variables représentant les paramètres de chaque composant puis d'écrire une boucle de traitement de gestion centrale testant les valeurs lues et agissant en fonction du résultat des tests. On notera qu'il y a d'un côté les variables et de l'autre, les actions.

2. La transition vers l'approche objet

La programmation objet est un paradigme de programmation considérant les différents acteurs d'un système comme des objets actifs et en relation. L'approche objet est souvent très voisine de la réalité.

Dans notre exemple, l'utilisateur ouvre le robinet ; le robinet relâche la pression et l'eau s'écoule du réservoir à l'évier ; le capteur/flotteur du réservoir arrive à un niveau qui déclenche la pompe ; l'utilisateur referme le robinet ; alimenté par la pompe, le réservoir du château d'eau se remplit et le capteur/flotteur atteint un niveau qui arrête la pompe.

Dans cette approche, vous constatez que les objets interagissent ; il n'existe pas de traitement central définissant dynamiquement le fonctionnement des objets. Il y a eu, en amont, une analyse fonctionnelle qui a conduit à la création des différents objets, à leurs montages et à leurs mises en relations.

Le code du programme "objet" va suivre cette réalité en proposant autant d'objets que décrit précédemment mais en définissant entre ces objets des méthodes d'échange adéquates qui conduiront au fonctionnement escompté.

Remarque

Les concepts objets sont très proches de la réalité...

3. Les caractéristiques de la POO

3.1 L'objet, la classe et la référence

3.1.1 L'objet

L'objet est l'élément de base de la POO. L'objet est l'union :

– d'une liste de variables d'états,

– d'une liste de comportements,

– d'une identification.

Les variables d'états changent durant la vie de l'objet. Prenons le cas d'un lecteur de musiques numériques. À l'achat de l'appareil, les états de cet objet pourraient être :

– mémoire libre = **100%**

– taux de charge de la batterie = **faible**

– aspect extérieur = **neuf**

Au fur et à mesure de son utilisation, ses états vont être modifiés. Rapidement la mémoire libre va chuter, le taux de charge de la batterie variera et l'aspect extérieur va changer en fonction du soin apporté par l'utilisateur.

Les comportements de l'objet définissent ce que peut faire cet objet : Jouer la musique - Aller à la piste suivante - Aller à la piste précédente - Monter le son, etc. Une partie des comportements de l'objet est accessible depuis l'extérieur de cet objet : bouton Play, bouton Stop... et une autre partie est uniquement interne : lecture de la carte mémoire, décodage de la musique à partir du fichier. On parle "d'encapsulation" pour définir une limite entre les comportements accessibles de l'extérieur et les comportements internes.

L'identification d'un objet est une information, détachée de la liste des états, permettant de différencier l'objet de ses congénères (c'est-à-dire d'autres objets qui ont le même type que lui). L'identification peut être un numéro de référence ou une chaîne de caractères unique construite à la création de l'objet ; elle peut également être une adresse mémoire. En réalité, sa forme dépend totalement de la problématique associée.

L'objet POO peut être attaché à une entité bien réelle, comme pour notre lecteur numérique, mais il peut être également attaché à une entité totalement virtuelle comme un compte client, l'entrée d'un répertoire téléphonique... La finalité de l'objet informatique est de gérer l'entité physique ou de l'émuler.

Même si ce n'est pas dans sa nature de base, l'objet peut être rendu persistant c'est-à-dire que ses états peuvent être enregistrés sur un support mémorisant l'information de façon intemporelle. À partir de cet enregistrement, l'objet pourra être recréé quand le système le souhaitera.

3.1.2 La classe

La classe est le "moule" à partir duquel l'objet va être créé en mémoire. La classe contient les états et les comportements communs d'un même type et les valeurs de ces états seront contenues dans ses objets.

Les comptes courants d'une même banque contiennent tous les mêmes paramètres (coordonnées du détenteur, solde...) et ont tous les mêmes fonctions (créditer, débiter...). Ces définitions doivent être contenues dans une classe et, à chaque fois qu'un client ouvre un nouveau compte, cette classe servira de modèle à la création de l'objet compte.

Le présentoir de lecteurs de musiques numériques propose un même modèle en différentes couleurs, avec des tailles mémoire modulables, etc. Chaque appareil du présentoir est un objet qui a été fabriqué à partir des informations d'une seule classe. À la réalisation, les attributs de l'appareil ont été choisis en fonction de critères esthétiques et commerciaux.

Une classe peut contenir beaucoup d'attributs. Ils peuvent être de type primitif – des entiers, des caractères... – mais également de type plus complexe. En effet, une classe peut contenir une ou plusieurs classes d'autres types. On parle alors de composition ou encore de "couplage fort". La destruction de la classe principale entraîne, évidemment, la destruction des classes qu'elle contient. Par exemple, si une classe *hôtel* contient une liste de *chambres*, la destruction de l'*hôtel* entraîne la destruction des *chambres*.

Une classe peut faire "référence" à une autre classe ; dans ce cas, le couplage est dit "faible" et les objets peuvent vivre indépendamment. Par exemple, votre PC est relié à votre imprimante. Si votre PC rend l'âme, votre imprimante fonctionnera certainement avec le futur PC ! On parle d'association.

En plus de ses attributs, la classe contient également une série de "comportements", c'est-à-dire une série de méthodes avec signatures et code attaché. Ces méthodes sont directement "copiées" dans les objets et utilisées telles quelles.

On déclare la classe et son contenu dans un même fichier source à l'aide d'une syntaxe que l'on étudiera en détail. Les développeurs C++ apprécieront le fait qu'il n'existe plus, d'un côté, une partie définitions et, de l'autre, une partie implémentations. En effet, en C#, le fichier programme (d'extension .CS) contient les définitions de tous les états avec éventuellement une valeur "de départ" et les définitions et implémentations de tous les comportements. On verra qu'il existe en C# une solution permettant de définir une classe dans plusieurs fichiers afin que des intervenants puissent travailler en parallèle sans jamais effacer le travail de l'autre.

3.1.3 La référence

Les objets sont construits à partir de la classe, par un processus appelé l'instanciation, et donc, tout objet est l'instance d'une classe. Chaque instance commence à un emplacement mémoire unique. Cet emplacement mémoire connu sous le nom de *pointeur* par les développeurs C et C++ devient une *référence* pour les développeurs C# et Java.

Quand le développeur a besoin d'un objet pendant un traitement, il doit :

– déclarer et nommer une variable du type de la classe à utiliser ;

– instancier l'objet et enregistrer sa référence dans cette variable.

Une fois cette instanciation réalisée, le programme accédera aux propriétés et aux méthodes de l'objet par la variable contenant sa référence. Chaque instance est unique. Par contre, plusieurs variables peuvent "pointer" sur une même instance. C'est d'ailleurs quand plus aucune variable ne pointe sur une instance donnée que le ramasse-miettes enregistre cette instance comme devant être détruite.

3.2 L'encapsulation

L'encapsulation consiste à créer une sorte de boîte noire contenant en interne un mécanisme protégé et en externe un ensemble de commandes qui vont permettre de le manipuler. Ce jeu de commandes est fait de telle sorte qu'il sera impossible d'altérer le mécanisme protégé en cas de mauvaise utilisation. La boîte noire sera si opaque qu'il sera impossible à l'utilisateur d'intervenir en direct sur le mécanisme.

Vous l'aurez compris, la boîte noire n'est autre qu'un objet avec des méthodes publiques de "haut niveau" contrôlant avec rigueur les paramètres passés avant de les utiliser dans un traitement. En respectant le principe de l'encapsulation, vous ferez en sorte que jamais l'utilisateur de l'objet ne puisse accéder "en direct" à vos données. Grâce à ce contrôle, votre objet sera correctement utilisé, donc plus fiable, et sa mise au point sera plus facile. Dans le monde Java, les méthodes permettant d'accéder en lecture aux données sont des accesseurs et celles permettant d'accéder en écriture sont des mutateurs. Nous verrons que C# propose une solution très élégante, les propriétés, qui permet de garder le côté pratique de l'accès direct aux données de l'objet tout en respectant les principes de l'encapsulation.

3.3 L'héritage

Une autre notion importante de la POO concerne l'héritage. Pour l'expliquer, imaginons que nous devions construire un système de gestion des différents types de collaborateurs d'une société. Une analyse rapide met en évidence une liste de propriétés communes à tous les postes. En effet, que le collaborateur soit ouvrier, cadre, intérimaire ou encore directeur, il a toujours un nom, un prénom et un numéro de sécurité sociale... On appelle cela la généralisation ; elle consiste à factoriser les éléments communs d'un ensemble de classes dans une classe plus générale appelée superclasse en Java et plutôt "classe de base" en C# et C++. La classe qui hérite de la superclasse est appelée sous-classe ou classe héritière.