Chapitre 3
Introduction au langage C#

1. La syntaxe

1.1 Les identifiants

Les identifiants sont les noms donnés aux classes et à leurs membres. Un identifiant doit être composé d'un seul mot commençant par une lettre ou un caractère underscore (_). Les identifiants peuvent être composés de lettres majuscules ou minuscules, mais le langage C# étant sensible à la casse, les majuscules et minuscules doivent être respectées pour faire référence au bon identifiant : `monIdentifiant` est différent de `MonIdentifiant`.

1.2 Les mots-clés

Les mots-clés sont des noms réservés par le langage C#. Ils sont interprétés par le compilateur et ne peuvent donc pas être utilisés en tant qu'identifiants. Ces mots-clés sont distingués dans l'éditeur de texte de Visual Studio en étant colorés en bleu (avec les paramètres d'apparence par défaut).

Si vous avez besoin d'utiliser un mot-clé en tant qu'identifiant pour un membre, il faut préfixer le nom de l'identifiant par le caractère @. La syntaxe suivante entraînera une erreur et le compilateur refusera de s'exécuter :

```
private bool lock;
```

En préfixant le membre `lock` par le caractère @, le compilateur considère que c'est un identifiant et non plus un mot-clé :

```
private bool @lock;
```

Le caractère @ peut également préfixer des identifiants qui n'ont aucun conflit avec les mots-clés, ainsi `@monIdentifiant` sera interprété de la même manière que `monIdentifiant`.

Voici une liste des mots-clés du langage C#. Ils seront expliqués, en partie, au cours de l'ouvrage :

abstract	add	as	ascending	async
await	base	bool	break	by
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	descending	do	double	dynamic
else	equals	enum	event	explicit
extern	false	file	finally	fixed
float	for	foreach	from	get
global	goto	group	if	implicit
in	int	interface	internal	into
is	join	let	lock	long
nameof	namespace	new	null	object
on	operator	orderby	out	override
params	partial	private	protected	public
readonly	ref	remove	required	return
sbyte	sealed	select	set	short

sizeof	stackalloc	statics	string	struct
switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe
ushort	using	value	var	virtual
volatile	void	where	while	yield

1.3 La ponctuation

La ponctuation a pour objectif de séparer les instructions du programme de manière logique, compréhensible par l'humain et interprétable par le compilateur.

Toute instruction doit se terminer par un point-virgule `;`. S'il est oublié à la fin de l'instruction, le compilateur lève une erreur de syntaxe. L'avantage, cependant, est de pouvoir écrire une instruction sur plusieurs lignes :

```
int i = 5 + 2;
```

Le point `.` après un identifiant permet d'accéder aux membres d'un objet. Visual Studio affiche, grâce à l'IntelliSense, la liste des membres disponibles dès que le point est ajouté à un objet :

```
monObjet.maPropriete
```

Les accolades `{` et `}` sont utilisées pour grouper plusieurs instructions au sein d'un bloc de contrôle ou d'une méthode. Elles indiquent à quel endroit les instructions commencent et à quel endroit elles se terminent :

```
class Program
{
}
```

Les parenthèses `(` et `)` sont utilisées pour la déclaration ou l'appel de méthodes. Elles peuvent contenir des paramètres suivant la signature de la méthode. Les paramètres d'une méthode sont séparés par une virgule `,` :

```
monObjet.maMethode(Parametre1, Parametre2);
```

Les parenthèses sont également utilisées pour grouper les instructions de la même manière que pour une opération mathématique.

Les crochets `[` et `]` permettent d'accéder à l'élément d'un tableau ou, si la classe contient une propriété indexeur, à l'élément d'une classe. Par exemple, si la classe `monObjet` est un tableau de valeurs de type `string`, pour accéder à son premier élément, la syntaxe serait la suivante :

```
string s = monObjet[0];
```

Les éléments des tableaux sont indexés à partir de 0.

1.4 Les opérateurs

1.4.1 Les opérateurs de calcul

Les opérateurs de calcul permettent, comme en mathématiques, d'effectuer des opérations.

L'addition est réalisée avec l'opérateur `+` :

```
i = 5 + 2;          // i = 7
```

La soustraction est réalisée avec l'opérateur `-` :

```
i = 5 - 2;          // i = 3
```

La multiplication est réalisée avec l'opérateur `*` :

```
i = 5 * 2;          // i = 10
```

La division est réalisée avec l'opérateur `/` :

```
i = 6 / 2;          // i = 3
```

Le modulo est réalisé avec l'opérateur `%` :

```
i = 5 % 2;          // i = 1
```

1.4.2 Les opérateurs d'assignation

Les opérateurs d'assignation permettent d'assigner une valeur à une variable. L'opérateur le plus utilisé est le caractère = :

```
i = x;
```

Il est également possible de réaliser une assignation et un calcul en même temps en combinant deux opérateurs :

```
i += 1;
```

En utilisant l'opérateur +=, il y a affectation à la variable de sa propre valeur additionnée de la valeur à droite de l'opérateur. Cette instruction est équivalente à celle-ci :

```
i = i + 1;
```

La combinaison d'opérateurs de calcul et d'assignation est valable pour tous les opérateurs :

```
int i = 5;
i += 2;          // i = 7
i -= 2;          // i = 5
i *= 2;          // i = 10
i /= 2;          // i = 5
i %= 2;          // i = 1
```

1.4.3 Les opérateurs de comparaison

Les opérateurs de comparaison sont essentiellement utilisés dans le cadre de décisions au sein d'instructions de contrôle.

L'opérateur == détermine si deux variables sont égales :

```
x == y           // renvoie true si x égale y
```

L'opérateur != détermine si deux variables sont différentes :

```
x != y           // renvoie true si x est différent de y
```

L'opérateur > détermine si la variable de gauche est strictement supérieure à la variable de droite :

```
x > y           // renvoie true si x est supérieur à y
```

L'opérateur `>=` détermine si la variable de gauche est supérieure ou égale à la variable de droite :

```
x >= y          // renvoie true si x est supérieur ou égal à y
```

L'opérateur `<` détermine si la variable de gauche est strictement inférieure à la variable de droite :

```
x < y          // renvoie true si x est inférieur à y
```

L'opérateur `<=` détermine si la variable de gauche est inférieure ou égale à la variable de droite :

```
x <= y          // renvoie true si x est inférieur ou égal à y
```

L'opérateur (et mot-clé) `is` permet de déterminer le type d'un objet :

```
x is int          // renvoie true si x est du type int
```

Le filtrage par motif permet de vérifier si une valeur correspond à un motif. Il s'agit d'utiliser l'opérateur `is` pour définir le motif qui peut être placé dans des instructions conditionnelles :

```
o is DateTime d   // renvoie true si o est du type DateTime
```

La variable `o` est automatiquement convertie dans le type testé et placée dans la nouvelle variable `d` utilisable de manière classique.

Il est également possible de combiner les opérateurs de comparaison avec des opérateurs logiques. L'opérateur `&&` permet de spécifier un ET logique tandis que l'opérateur `||` spécifie un OU logique. Les différentes expressions peuvent être combinées grâce à des parenthèses afin de modifier l'ordre d'interprétation :

```
(x >= 0 || x > 10) && (x <= 1 || x < 25)
```

1.5 La déclaration de variables

La déclaration de variables se fait en spécifiant son type puis en indiquant son identifiant. L'instruction suivante déclare une variable nommée `s` et du type `string` :

```
string s;
```

L'instruction de déclaration se termine comme toutes les instructions par le point-virgule.

Une variable peut être déclarée et initialisée avec la même instruction :

```
string s = "La valeur de ma variable";
```

Il est également possible de déclarer et d'initialiser plusieurs variables en une seule instruction, à la condition qu'elles soient de type identique. Les variables sont séparées par une virgule :

```
bool b1 = true, b2 = false;
```

Une variable peut également être marquée avec le mot-clé `const` qui spécifie que la valeur de la variable ne peut pas être modifiée pendant l'exécution. C'est une variable en lecture seule :

```
const int i = 0;
```

En ajoutant le mot-clé `required` à une propriété ou à un champ, vous indiquez l'obligation pour les appelants du constructeur d'initialiser ces valeurs.

```
required int i;
```

La portée d'une variable déclarée dans le corps d'une méthode se limite à celle-ci, c'est-à-dire qu'elle est détruite dès la fin de l'exécution de la méthode. Elle sort de la portée.

Les variables déclarées au sein d'un bloc conditionnel ou itératif ne sont accessibles que dans ce bloc. Pour les blocs itératifs, à la fin de la boucle la variable est détruite puis elle est à nouveau initialisée au passage suivant.

1.6 Les instructions de contrôle

1.6.1 Les instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter une portion de code en fonction de tests effectués sur les variables de l'application. Il existe deux types de structures conditionnelles : les blocs `if` et les blocs `switch`.

if, else et else if

Syntaxe générale

```
if (expression)
{
       instructions
}
[else if (expression)
{
       instructions
}]
[else
{
       instructions
}]
```

L'instruction `if` évalue une expression booléenne et exécute le code si cette expression est vraie (`true`). Sa syntaxe est la suivante :

```
if (x > 10)
{
    // Instructions exécutées si x est supérieur à 10
}
```

L'expression à évaluer est insérée entre parenthèses après le mot-clé `if` et doit avoir un résultat de type booléen.

L'instruction `else` permet d'intégrer le code qui sera exécuté si l'expression évaluée dans l'instruction `if` est fausse (`false`) :

```
if (x > 10)
{
   // Instructions exécutées si x est supérieur à 10
}
else
{
   // Instructions exécutées si x est inférieur ou égal à 10
}
```