Chapitre 4
Algorithmique

1. Bases de l'algorithmique

Jusqu'à présent, nous nous sommes contentés de développer des applications n'incluant aucune "logique" : elles se limitaient à afficher des données. Il n'y avait aucune notion de condition, de répétition ou même de logique de code. En effet, le code d'une application est souvent complexe et les embranchements sont multiples en fonction de diverses conditions. Dans ce chapitre, nous allons découvrir la logique algorithmique, qui vous permettra de créer du code plus proche de ce que l'on peut retrouver dans les applications répondant à des problématiques plus complexes.

1.1 La logique conditionnelle

Indéniablement, il s'agit ici d'une brique que vous allez utiliser de façon systématique. Une condition implique l'exécution ou non d'une partie du code en fonction de l'évaluation d'un test logique.

1.1.1 Test simple : le if/else

La logique conditionnelle se traduit en pseudocode de la façon suivante :

```
SI une condition ALORS
  Je fais quelque chose
SINON
  Je fais autre chose
```

En C#, les mots-clés pour réaliser une instruction conditionnelle sont `if` et `else` :

```
if(condition)
{
    ....
}
else
{
    ....
}
```

La condition testée par une instruction `if` doit renvoyer un booléen. Ce dernier peut être stocké dans une variable mais il est également possible que l'instruction `if` évalue directement la condition, sans variable intermédiaire.

Si on reprend l'exemple de la fin du chapitre précédent, on pourrait améliorer notre classe `Voiture` pour rajouter un booléen qui indique si l'instance de la voiture est fonctionnelle. Si la valeur est égale à "oui", il est inutile de réparer la voiture. Cependant, si la voiture n'est pas fonctionnelle, il faut la réparer :

```
public class Voiture
{
    public bool Fonctionnelle { get; set; }
    ...
}
public class Garage
{
    public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle)
        {
              Console.WriteLine("La voiture n'a pas besoin d'être
réparée car elle est fonctionnelle");
        }
```

```
        else
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
}
```

Comme on le voit dans le code ci-dessus, l'instruction `if` se base sur la valeur booléenne stockée dans la propriété `Fonctionnelle` de la classe `voiture` pour évaluer si oui ou non la réparation est nécessaire. Ici, le test a été fait de telle sorte que l'on vérifie si la condition est vraie, et dans le cas inverse, on effectue la réparation. On peut très bien inverser la condition initiale, en comparant le booléen à la valeur `false`. De ce fait, on peut même se passer du `else`, qui n'apporte pas réellement de plus-value :

```
public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle == false)
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
```

À noter également qu'il est possible d'inverser la valeur d'un booléen en mettant un point d'exclamation en préfixe. Ainsi, `!true` est égal à `false`, et `!false` est égal à `true`. Même si cela peut sembler compliqué de prime abord, vous verrez que c'est une façon d'écrire qui deviendra rapidement automatique à l'utilisation. Si l'on reprend l'exemple précédent, le code qui utilise l'inversion de valeur avec le point d'exclamation serait le suivant :

```
public void Repare(Voiture voiture)
    {
        if(!voiture.Fonctionnelle)
        {
              Console.WriteLine("Réparation de la voiture");
              voiture.Fonctionnelle = true;
        }
    }
```

Même si à première vue l'instruction `else` est utilisée pour définir le cas inverse de celui du `if` principal, elle peut également servir de base pour une autre instruction `if` à suivre afin de faire une instruction ayant pour sémantique "sinon si". Il suffit dans ce cas d'ajouter une condition `if` après le `else`. Par exemple :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
         Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

À noter que la structure du code conditionnel est très flexible : on peut avoir uniquement une seule instruction `if`, une instruction `if` et son `else` associé, ou un enchaînement de `if` et `else if` (avec ou sans `else` final). La seule impossibilité : avoir une instruction `else` seule, car cette dernière indique forcément l'inverse d'une condition donnée.

Remarque

Pour que ces embranchements soient possibles, il faut bien sûr qu'il y ait des conditions pouvant donner plusieurs résultats. À ce titre, il n'est pas utile de faire un `if`, `else if`, `else` avec un simple booléen, car ce dernier ne pouvant avoir que deux états, un `if` avec un `else` est suffisant.

Une instruction `if` peut être "compressée" en l'exprimant sous une forme réduite appelée ternaire. Généralement, on utilise cette approche afin d'écrire en ligne un test pour éviter une lourdeur syntaxique, et ce afin d'affecter le contenu d'une variable. La syntaxe est la suivante : on définit en première partie le test à évaluer, séparant d'un point d'interrogation le test des résultats. Ensuite, les cas vrai et faux sont tous deux séparés par un deux-points. La syntaxe est la suivante :

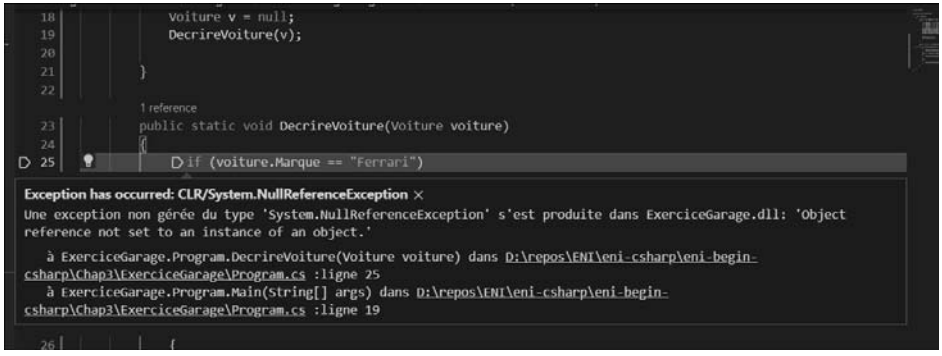
```
test ? cas si vrai : cas si faux
```

Par exemple :

```
string voiture = voiture.Marque == "Ferrari" ? "Voiture chère" :
"Voiture peu chère";
```

Il est possible d'enchaîner les ternaires avec l'usage des parenthèses (en refaisant une autre ternaire dans un cas ou l'autre) mais il est recommandé d'agir avec parcimonie afin de conserver une lisibilité de code optimale.

Il est souvent intéressant de tester si un objet a été affecté avant d'accéder à ses données ou à ses méthodes. En l'absence de ce test, cela peut provoquer une erreur à l'exécution (appelée exception, que nous détaillerons dans ce chapitre à la section La gestion des erreurs). Si on reprend le code précédent, étant donné que `Voiture` est une classe, elle peut donc valoir la valeur `null`. La fonction `DecrireVoiture` tenterait dès lors d'accéder à une variable qui n'a pas de valeur, provoquant une erreur à l'exécution :



Erreur à l'exécution

Afin d'effectuer un quelconque test sur une donnée d'une classe ou de faire un appel de méthode, il est recommandé de tester si la valeur est bien différente de `null`. Cela peut se faire de façon "classique" ou grâce au nouvel apport du mot-clé `not` de C# 9 (comme cela est décrit dans la section à venir Pattern matching) :

```
public void DecrireVoiture(Voiture voiture)
{
    if (voiture != null) // avant C# 9
    {
          ...
    }
    if (voiture is not null) // depuis C# 9
    {
          ...
    }
}
```

Pour éviter ce genre de problèmes, un opérateur de navigation sécurisé a été ajouté en C# 6. Ce dernier permet de n'accéder à une méthode ou de lire une donnée que si la variable n'est pas `null`. On utilise le point d'interrogation juste après la variable, avant l'appel, et cela permet de se passer de tester la nullité :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture?.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture?.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
         Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

Le fonctionnement de cet opérateur est le suivant :

- Si la variable n'est pas `null`, on accède à la propriété ou à la méthode concernée normalement.
- Si la variable est `null` :
  - S'il s'agit d'un appel d'une méthode, et que la méthode ne renvoie rien, elle ne sera pas invoquée ;
  - S'il s'agit d'un appel d'une méthode et que la méthode renvoie une valeur, ou qu'il s'agit d'un appel à une propriété, il faudrait tester si la valeur est différente de `null`. S'il s'agit d'un type référence (d'une classe, comme un `string`), alors il faudrait tester si c'est `null` ou pas, afin de voir si l'appel a été fait. S'il s'agit d'un type valeur, alors le type sera encadré d'un nullable. Par exemple, si le type de retour était un `int`, on obtiendrait un `int?` lors de l'appel, qui serait égal à `null` si la variable était à `null`, ou qui aurait la valeur le cas contraire.

```
public class TestClass
{
    public int Valeur { get; set; }
    public string ValeurString { get; set; }
    public void Methode() { }
    public int MethodeInt()
    {
        return 42;
    }
    public string MethodeString()
    {
        return "valeur";
    }
}

TestClass c = null;
int? valeur = c?.Valeur;
string valeurStr = c?.ValeurString;
c?.Methode();
int? retour = c?.MethodeInt();
string retourStr = c?.MethodeString();
```

Chapitre 4
Les types du C#

1. "En C#, tout est typé !"

Le terme générique "**type**" regroupe les classes, les structures, les records, les interfaces, les énumérations et les délégués. Ces types sont décrits dans la CTS (*Common Type System*) pour que des compilateurs de langages différents puissent générer un code exploitable par la CLR (*Common Language Runtime*). Un programme utilise les différents types et un assemblage peut implémenter plusieurs types.

Voici les définitions succinctes des différents types proposés par le C# :

– Le type "Classe" est l'implémentation C# de ce qui a été présenté dans les premiers chapitres. La classe est évidemment le type le plus utilisé dans les applications. Le chapitre Création de classes en définit précisément la syntaxe de déclaration, d'allocation et d'utilisation.

– Le type "Structure" est assez voisin de celui du langage C. Avant la démocratisation de la programmation objet, les structures étaient le moyen le plus commun offert aux développeurs pour composer leurs propres types. Retenons pour l'instant que les structures du C# sont très proches des classes et que, quand elles sont utilisées à bon escient, elles peuvent améliorer les performances d'une application. Nous verrons au chapitre suivant que le .NET encapsule la plupart de ses types "simples" (les entiers, les caractères, etc.) dans des structures. Sachez que les structures n'existent pas en Java.

– Le type "Record" est un objet pouvant être classe ou structure qui s'identifie par son contenu et non par son emplacement en mémoire. Cette distinction subtile sera détaillée plus loin.

– Le type "Interface" est largement utilisé dans le .NET et contribue à la communication entre les classes. Retenons pour l'instant qu'une interface est une classe souvent sans code qui formalise un lot de méthodes obligatoires pour la classe qui l'implémentera. Le chapitre Héritage et polymorphisme traite du sujet.

– Le type "Enumération" permet la définition de listes clés-valeurs et la création des données dont les contenus seront limités à ces clés. Rappelons l'exemple d'un type *Jour* pouvant contenir de *lundi* à *dimanche*. Si, lors de la rédaction du programme, on tente de copier dans un objet de ce type la clé *Mars*, il y aura une erreur de compilation. En C#, ce type apporte un lot de méthodes permettant de gérer cette liste par programmation.

– Le type "Delegate" (Délégué) encapsule la notion de pointeur de fonction du C/C++, origine de bien des soucis, en commençant par lui attribuer un typage fort. En effet, le pointeur de fonction "conventionnel" n'est autre qu'une adresse mémoire sans autre précision sur la signature et l'application s'arrête en erreur quand les paramètres passés ne correspondent pas aux paramètres attendus... C'est pourquoi le type *delegate* du C# va être défini précisément avec la signature de la méthode qui lui sera associée. Ensuite, l'instance de type *delegate*, généralement créée au sein d'une classe amenée à communiquer avec d'autres, gère une liste "d'abonnés" via une syntaxe déconcertante de simplicité. Il suffit en effet d'utiliser l'opérateur += du *delegate* pour s'abonner à la liste de diffusion et -= pour s'en désenregistrer. Les *delegate* sont très largement utilisés dans le C# ; on les retrouvera beaucoup dans les interfaces graphiques pour que les composants puissent notifier l'application de leurs changements d'états.

Durant ce chapitre seront abordées des notions illustrées par des extraits de code. Ces extraits de code utilisent des syntaxes décrites dans les chapitres suivants mais la compréhension des chapitres suivants passe... par celle de ce présent passage ! Vous avez donc à prendre pour argent comptant dans un premier temps les syntaxes des exemples mais nous les approfondirons par la suite.

Remarque

Tous les exemples de ce chapitre sont des projets de la solution Visual Studio TypesDuCSharp.sln figurant dans le répertoire Chap4 du .zip accompagnant cet ouvrage. Ce fichier d'accompagnement est à télécharger sur le site des Éditions ENI www.editions-eni.fr.

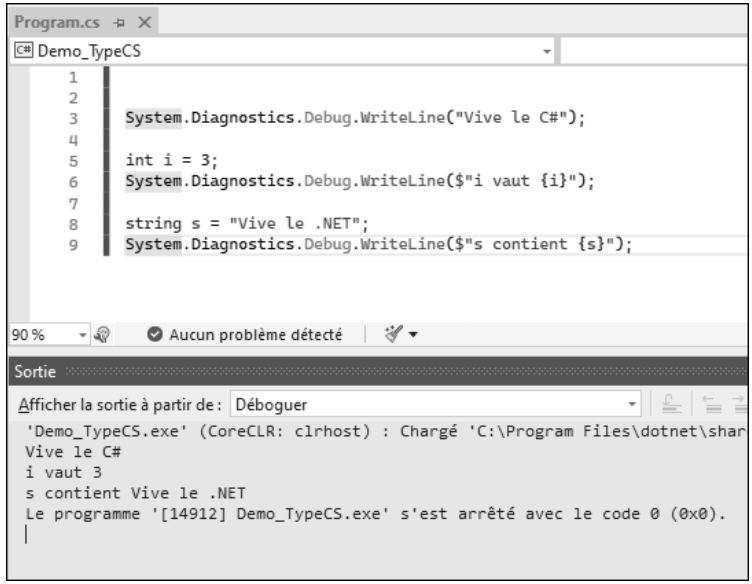
Introduction à System.Diagnostics.Debug

Ces exemples utilisent des classes de tests et également une classe système appelée *System.Diagnostics.Debug*. Cette classe permet, entre autres, d'écrire des messages dans la fenêtre **Sortie** de Visual Studio et également de vérifier des conditions passées en paramètres. Elle sera étudiée avec d'autres classes de même type dans le chapitre Traçage et instrumentation des applications.

Syntaxe d'affichage dans la fenêtre **Sortie** de Visual Studio

```
System.Diagnostics.Debug.WriteLine("le message");
```

Exemple d'utilisations simples et composées



Le signe $ qui précède le contenu de la chaîne permet d'effectuer une "interpolation", à savoir un remplacement de séquence {blabla} par le contenu de la variable blabla. Cette extension très souple et très pratique est arrivée avec le C# 6.

La méthode *System.Diagnostics.Debug.Assert* permet de vérifier qu'une condition est vraie pendant l'exécution de votre code. En utilisant cette méthode vous n'intervenez pas sur le déroulement du programme en tant que tel ; vous vérifiez juste que ce qui est prévu à tel endroit du code est correct. Si la condition est fausse, une boîte de dialogue sera affichée pour vous en informer.

Syntaxe d'utilisation de la méthode System.Diagnostics.Debug.Assert

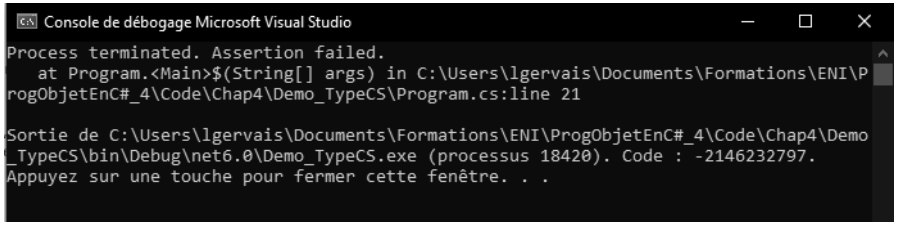
```
System.Diagnostics.Debug.Assert(<condition>);
```

Exemples d'utilisation de la méthode Assert

```
// Vérification de la condition "1 est différent de 2"
System.Diagnostics.Debug.Assert(1 != 2);
// Comme la condition est vraie, le programme
// passe à la ligne suivante


// Pour voir l'effet produit
// lorsqu'une condition n'est pas vérifiée,
// une erreur est "forcée" en ligne suivante
System.Diagnostics.Debug.Assert(1 == 2);
```

À l'exécution de la seconde ligne de l'extrait, le programme affiche une boîte de message et attend que l'utilisateur la referme avant de poursuivre l'exécution du code.

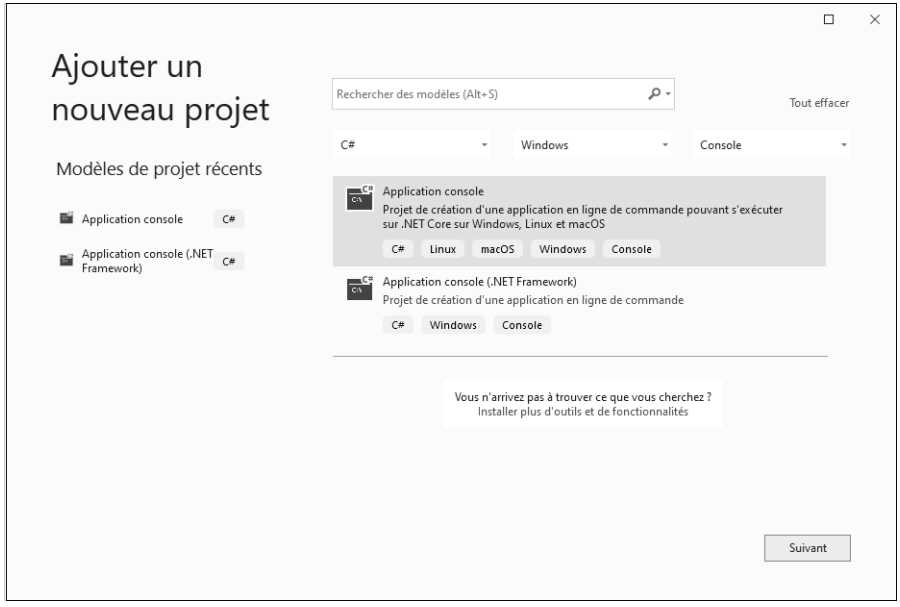


Ainsi, vous pouvez vérifier que ce que vous avez prévu se réalise correctement. On utilise *System.Diagnostics.Debug.Assert* principalement pendant les phases de mise au point pour éventuellement ajouter du code de protection par la suite. Nous verrons d'ailleurs que Visual Studio génère une version "mise au point" (*Debug*) et une version "production" (*Release*). *System.Diagnostics.Debug.Assert* n'a aucun effet sur un code compilé en mode "production".

Introduction à System.Console

Nous l'avons déjà utilisée au chapitre Introduction à .NET 6 et à VS pour afficher le classique *Hello World* à l'écran ; la console va servir de support à plusieurs exemples à suivre. Cet environnement d'exécution très sommaire présente l'avantage de pouvoir simplement afficher des chaînes à l'écran et lire des entrées clavier.

Comme nous l'avons vu, le type **Application console** se choisit à la création du projet.



Voici les principales commandes qui seront utilisées :

Affichage d'une chaîne suivie d'un changement de ligne

```
Console.WriteLine("Message à afficher...");
```

Affichage d'un type primitif sans changement de ligne

```
      int j = 358;
      Console.Write(j);
```

Affichage d'une composition

```
int k = 2;
int l = 3;
Console.WriteLine($"k contient {k} et l contient {l}");
```

Lecture d'une chaîne de caractères saisie au clavier
et terminée par la touche [Entrée]

```
string saisie = Console.ReadLine();
```

Ces présentations étant faites, nous pouvons passer à la suite...

2. "Tout le monde hérite de System.Object"

Le type *System.Object* est la base directe ou indirecte de tous les types du .NET, ceux existants et ceux que vous allez créer (la notion d'héritage a déjà été un peu abordée dans les premiers chapitres). L'héritage d'*Object* étant implicite, sa déclaration est inutile. Tous les types héritent de ses méthodes et peuvent même en substituer certaines.

C'est ce que fait *System.ValueType* qui, dans la hiérarchie des types du .NET, devient la base de la famille "Valeurs" en adaptant les méthodes de *System.Object*.

2.1 Les types Valeurs

La famille "Valeurs" se divise en plusieurs parties :

– les énumérations
– les structures
– les records (s'ils sont instanciés en type Valeur)

Les structures sont elles-mêmes sous-divisées en :

– types numériques :
  – les types intégraux :

Type	Taille
sbyte	Entier signé sur 8 bits
byte	Entier non signé sur 8 bits
char	Caractère UNICODE 16 bits
short	Entier signé sur 16 bits
ushort	Entier non signé sur 16 bits
int	Entier signé sur 32 bits
uint	Entier non signé sur 32 bits
long	Entier signé sur 64 bits
ulong	Entier non signé sur 64 bits

  – les types à virgule flottante :

Type	Précision
float	7 chiffres
double	15-16 chiffres

  – le type décimal (adapté aux calculs financiers) :

Type	Précision
decimal	28-29 chiffres