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 :

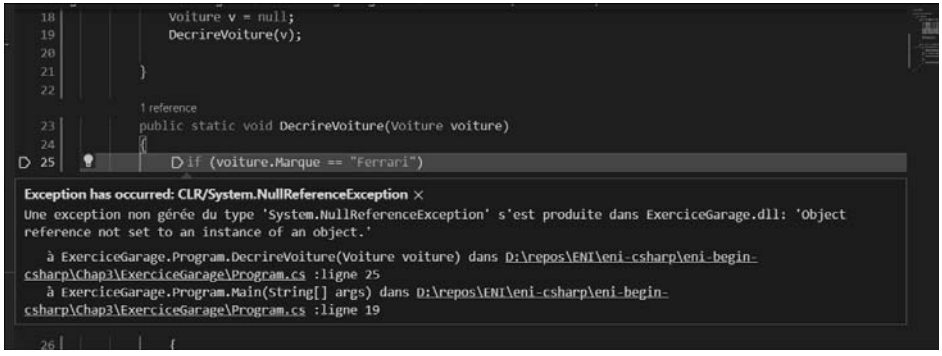
```
test ? cas si vrai : cas si faux
```

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
La création de types

1. Introduction

Les classes représentent la majorité des types référence. La définition la plus simple d'une classe sera :

```
class MaClasse
{
}
```

Au fur et à mesure de la construction d'une classe, des éléments sont ajoutés :

– Les membres (méthodes, propriétés, indexeurs, événements...) sont placés entre les accolades.

– Les attributs et les modificateurs de classe comme le niveau d'accès sont placés avant le mot-clé `class`.

– L'héritage et les implémentations d'interfaces sont placés après le nom de la classe.

2. Les niveaux d'accès

Les niveaux d'accès permettent de définir comment vont pouvoir s'effectuer l'instanciation des types et l'appel des méthodes. Le niveau d'accès est défini à l'aide de mots-clés précédant la déclaration de la classe, ou du membre. Le tableau suivant présente les modificateurs d'accès disponibles :

Modificateur d'accès	Description
`public`	Autorise l'accès pour tous les types de l'assemblage et hors de l'assemblage.
`private`	Autorise l'accès uniquement pour les autres membres du type.
`internal`	Autorise l'accès pour tous les types de l'assemblage uniquement.
`protected`	Autorise l'accès uniquement pour les autres membres du type ou pour les types héritant de celui-ci même en dehors de l'assemblage.
`protected internal`	Autorise l'accès uniquement pour les autres membres du type ou pour les types héritant de celui-ci dans l'assemblage uniquement.

Si aucun modificateur d'accès n'est précisé sur un membre, il est considéré comme `private`. Une classe ou une structure sans modificateur d'accès sera considérée comme `public`.

Les membres ne pourront jamais étendre leur niveau d'accès au-delà de celui du type contenant. Cela signifie que même si un membre est marqué avec le modificateur d'accès `public`, et que la classe dans laquelle il se trouve est marquée comme `internal`, le membre ne sera accessible que pour les types de l'assemblage :

```
internal class MaClasse
{
    // Le membre est accessible depuis l'assemblage uniquement
    public int i;
}
```

Même si des membres sont marqués comme `internal`, il est possible de les exposer à d'autres assemblages. Il suffit d'ajouter un attribut du type `System.Runtime.CompilerServices.InternalsVisibleTo` en spécifiant le nom de l'assemblage dans le fichier **AssemblyInfo.cs** comme ceci :

```
[assembly: InternalsVisibleTo("Assemblage")]
```

Si l'assemblage à autoriser est signé avec un nom fort, vous pouvez spécifier son nom complet :

```
[assembly: InternalsVisibleTo("Assemblage, Version=1.0.0.0,
                  Culture=fr, PublicKeyToken=26381116d3a4ad13")]
```

3. Les structures

Les structures sont très similaires aux classes avec, comme principales différences, le fait qu'une structure est un type valeur alors qu'une classe est un type référence. Les structures sont utilisées à la place des classes quand la sémantique exige un type valeur. Une structure ne supporte pas l'héritage. Elles peuvent posséder tous les membres d'une classe à l'exception d'un constructeur sans paramètre, d'un destructeur et de membres virtuels.

Voici l'exemple de la définition d'une structure :

```
struct GeoPoint
{
    double Longitude;
    double Latitude;
}
```

La déclaration et l'instanciation d'une occurrence de la structure se font comme pour une classe. Un constructeur sans paramètre existe implicitement :

```
GeoPoint g = new GeoPoint();
```

Il est possible de surcharger ce constructeur pour initialiser les membres de la structure mais il faut noter que, lors de l'utilisation du mot-clé `default`, l'initialisation affectera les valeurs par défaut des types aux membres (dans l'exemple ci-dessous, 0 pour les types `double`) :

```
public GeoPoint()
{
    this.Longitude = 1;
```

```
    this.Latitude = 2;
}
  var g1 = new GeoPoint();
  // g1.Longitude = 1
  // g1.Latitude = 2

  var g2 = default(GeoPoint);
  // g2.Longitude = 0
  // g2.Latitude = 0
```

Il est possible de rajouter son propre constructeur à partir du moment où tous les champs de la structure sont initialisés :

```
Public GeoPoint(double longitude, double latitude)
{
    this.Longitude = longitude;
    this.Latitude = latitude;
}
```

Les champs peuvent être initialisés lors de leur déclaration dans la structure :

```
struct GeoPoint
{
    double Longitude = 1;
    double Latitude;
}
```

4. Les classes

4.1 Les champs

Un champ est une variable qui est un membre de la classe. Il peut s'agir de type valeur ou de type référence.

À la racine du projet, créez un dossier nommé **Library** et créez une nouvelle classe nommée `Project`. Ajoutez les champs suivants :

```
public class Project
{
    protected string filename = "sans titre.smpx", path;
    protected DataTable data = new DataTable();
```

```
    protected bool hasChanged;
}
```

Les champs peuvent être initialisés au moment de la déclaration. Un champ qui n'est pas initialisé explicitement recevra les valeurs par défaut suivant son type. L'initialisation des champs est effectuée avant l'exécution du constructeur de la classe.

Il est également possible de déclarer et initialiser plusieurs champs en une seule instruction s'ils ont le même niveau d'accès et le même type :

```
private string filename = "sans titre.smpx", path;
```

Le mot-clé `readonly` permet de spécifier qu'un champ sera en lecture seule, il pourra seulement être assigné lors de la déclaration ou lors de l'instanciation, au sein du constructeur :

```
public readonly int i = 1;
```

4.2 Les propriétés

Les propriétés ressemblent à des champs puisqu'on y accède de la même manière mais leur logique interne les rapproche des méthodes.

Une propriété est déclarée de la même manière qu'un champ en ajoutant des blocs `get` et `set`. Ces deux blocs sont appelés des accesseurs ; l'accesseur `get` est exécuté lorsque la propriété est lue et doit retourner une valeur du type de la propriété. L'accesseur `set` est exécuté lorsque la propriété est assignée. Un paramètre implicite accessible via le mot-clé `value` du type de la propriété est fourni.

Utilisez les outils de refactorisation de Visual Studio ([Ctrl] **R**, **E** avec le curseur sur le nom d'un champ) pour générer les propriétés des champs précédemment créés dans la classe `Project`. Le code généré est le suivant :

```
public string Filename
{
    get { return filename; }
    set { filename = value; }
}
public string Path
{
```

```
    get { return path; }
    set { path = value; }
}
public DataTable Data
{
    get { return data; }
    set { data = value; }
}
public bool HasChanged
{
    get { return hasChanged; }
    set { hasChanged = value; }
}
```

Il est possible de créer des propriétés automatiques au lieu de créer un champ puis une propriété avec des accesseurs qui ont pour seul but de lire et écrire dans un champ privé :

```
public int i { get; set; }
```

Ce type de déclaration indique au compilateur de générer automatiquement un champ privé de la propriété qui lui servira pour stocker les valeurs.

Depuis la version 6 de C#, il est possible de spécifier uniquement un accesseur `get` dans la déclaration de la propriété. Le compilateur crée alors un champ privé en lecture seule. Il est également possible d'initialiser la propriété directement sans passer par un constructeur :

```
public int i { get; } = 10;
```

Les accesseurs peuvent être marqués avec différents niveaux d'accès. Ainsi, l'accesseur `set` pourra être marqué par le mot-clé `private` afin d'exposer la propriété en lecture seule :

```
public int i { get; private set; }
```

Un accesseur possède, par défaut, le même niveau d'accès que celui qui marque la propriété.

Il est possible de créer une propriété immuable (dont la valeur est fixée à l'instanciation de l'objet) en remplaçant le mot-clé `set` par le mot-clé `init`. Cela aura pour effet de pouvoir éviter l'écriture d'un constructeur avec la syntaxe suivante :

```
MyObject o = new() { i = 10 } ;
```

Vous pouvez utiliser un modèle de propriété pour déterminer sa valeur en fonction de celle des membres imbriqués :

```
  static bool ValeurNonZero(GeoPoint g) => g is not { X: 0, Y: 0 };
```

Et également en fonction des propriétés des membres imbriqués :

```
  public struct Position
  {
      public GeoPoint Longitude;
      public GeoPoint Latitude;
  }
  static bool ValeurNonZero(Position p) => p is not
    { Longitude.X: 0, Longitude.Y: 0 }
    and
    { Latitude.X: 0, Latitude.Y: 0 };
```

La classe `Project` contient une propriété `HasChanged` de type booléen. Elle doit refléter le fait que l'objet a été modifié ou non depuis la dernière sauvegarde. Pour ce faire, ajoutez le code permettant de mettre à jour le champ `HasChanged` lorsque les propriétés `Filename`, `Path` et `Data` sont modifiées. Voici l'exemple avec la propriété `Filename` :

```
public string Filename
{
    get { return filename; }
    protected set
    {
        if (this.filename != value)
        {
            this.filename = value;
            this.HasChanged = true;
        }
    }
}
```