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 :

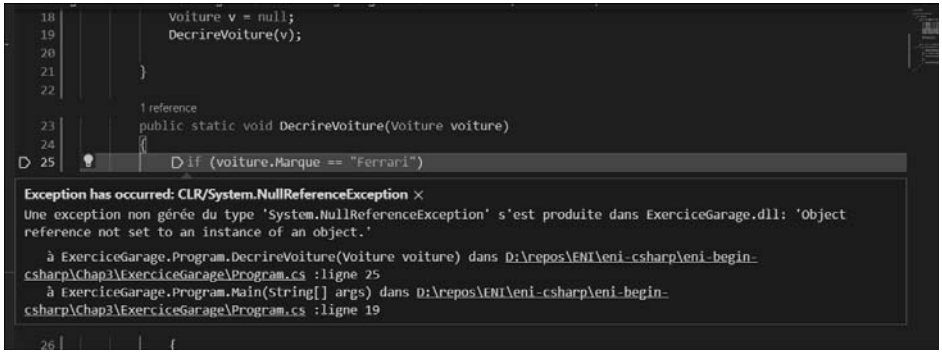
```
test ? cas si vrai : cas si faux
```

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 3
Les ordres du SQL

1. Les bases du langage SQL

1.1 Les expressions

Dans la plupart des syntaxes Transact-SQL, il est possible d'utiliser des expressions ou des combinaisons d'expressions pour gérer des valeurs ou pour tirer parti de la programmabilité du langage. Les expressions peuvent prendre différentes formes :

Les valeurs constantes

Exemple

Caractère `'AZERTY', 'Ecole Nantaise d''Informatique'`

Les chaînes de caractères sont entourées d'apostrophes. S'il doit y avoir une apostrophe dans la chaîne de caractères, il est nécessaire d'en mettre deux pour ne pas le confondre avec la fin de la chaîne de caractères.

```
Numérique       10, -15.26, 1.235e-5
```

Les valeurs numériques peuvent être écrites avec des chiffres en utilisant le point pour séparer la partie entière de la partie décimale. Il est possible d'utiliser la notation scientifique en utilisant le ***e*** avant la puissance de :

```
Date
constante              date                heure
'231205'               5 Décembre 2023     00:00:00.000
'05/12/2023'           5 Décembre 2023     00:00:00.000
'05-12-23 8:30'        5 Décembre 2023     08:30:00:000
'8:30:2'               1 Janvier 1900      08:30:02:000
'5.12.2023 8:30pm'     5 Décembre 2023     20:30:00:000
```

Le format français jj/mm/aaaa est l'un des formats possibles pour les dates.

```
Binaire         0x05,  0xFF,  0x5aef1b
Nulle           NULL
```

Les données binaires doivent être données en hexadécimal et sont préfixées par 0x.

Les noms de colonne

Un nom de colonne pourra être employé comme expression, la valeur de l'expression étant la valeur "stockée" de la colonne.

Les fonctions

On peut utiliser comme expression n'importe quelle fonction, la valeur de l'expression est le résultat retourné par la fonction.

Exemple

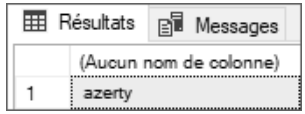
```
expression                      valeur
SQRT(9)                         3
substring('ABCDEF',2,3)         'BCD'
```

Les variables

Les variables peuvent être employées en tant qu'expression ou dans une expression, sous la forme @nom_de_variable. La valeur de l'expression est la valeur de la variable.

Exemple

```
DECLARE @x CHAR(10);
SELECT @x='AZERTY';
SELECT LOWER(@x);
```



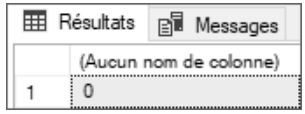
Les sous-requêtes

Une requête SELECT entre parenthèses peut être employée en tant qu'expression ayant pour valeur le résultat de la requête, soit une valeur unique, soit un ensemble de valeurs. Les requêtes SELECT sont présentées à la section L'extraction de lignes de ce chapitre.

Exemple

Stockage du nombre de clients dans une variable :

```
DECLARE @nombre INT;
SELECT @nombre = (SELECT COUNT(*) FROM Clients);
SELECT @nombre;
```



Les expressions booléennes

Elles sont destinées à tester des conditions (IF, WHILE, WHERE, etc.). Ces expressions sont composées de la manière suivante :

```
expression1 opérateur expression2
```

1.2 Les opérateurs

Les opérateurs vont permettre de constituer des expressions calculées, des expressions booléennes ou des combinaisons d'expressions.

1.2.1 Les opérateurs arithmétiques

Ils permettent d'effectuer des calculs élémentaires et de renvoyer un résultat.

+ Addition

- Soustraction

* Multiplication

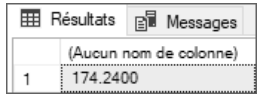
/ Division : division entière si les deux opérandes sont des entiers, division réelle sinon.

% Modulo (reste de la division entière)

(...) Parenthèses

Exemple

```
DECLARE @prixArticle NUMERIC(6,2) = 145.2;
DECLARE @TVA NUMERIC(2,2) = 20.0/100;
SELECT @prixArticle*(1+@TVA);
```




```
DECLARE @nb INT = (SELECT COUNT(*) FROM Clients);
IF @nb%2=0 PRINT 'nombre de clients pair';
ELSE PRINT 'nombre de clients impair';

nombre de clients pair
```

Remarque

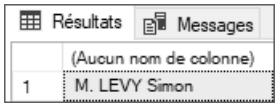
Les opérateurs + et - fonctionnent également sur les dates. L'unité est alors le jour. Exemple : `GETDATE()+1` permet d'obtenir la date du lendemain.

1.2.2 La concaténation de chaînes de caractères

La concaténation permet de constituer une seule chaîne de caractères à partir de plusieurs expressions de type caractère. L'opérateur de concaténation est le signe plus (+).

Exemple

```
DECLARE @nom CHAR(4) = 'LEVY';
DECLARE @prenom CHAR(5) = 'Simon';
SELECT 'M. ' + @nom + ' ' + @prenom;
```



Il existe de plus la fonction CONCAT() qui permet de réaliser des concaténations. Contrairement à l'opérateur +, cette fonction appartient à la norme SQL. Elle a également l'avantage de pouvoir être utilisée avec des nombres sans que la somme soit effectuée.

1.2.3 Les opérateurs binaires

Ils permettent le calcul entre entiers, traduits implicitement en valeurs binaires.

& ET

| OU

^ OU Exclusif

~ NON

1.2.4 Les opérateurs de comparaison

Ils permettent la constitution d'expressions booléennes en comparant des expressions. Ces expressions peuvent être placées entre parenthèses.

`exp1 = exp2`

Égal.

`exp1 > exp2`

Supérieur.

`exp1 >= exp2`

Supérieur ou égal.

Remarque

`exp1!>exp2` est également possible mais n'est pas dans la norme.

`exp1 < exp2`

Inférieur.

`exp1 <= exp2`

Inférieur ou égal.

Remarque

`exp1 !< exp2` est également possible mais n'est pas dans la norme.

`exp1 <> exp2`

Différent.

Remarque

`exp1 != exp2` est également possible mais n'est pas dans la norme.

`exp IN (exp1, exp2,...)`

Indique si la valeur est parmi celles de la liste de valeurs.

`exp IS NULL` ou `exp IS NOT NULL`

Teste l'absence de valeur (NULL). Pour tester si une variable n'a pas de valeur affectée, il est indispensable d'utiliser l'opérateur IS NULL.

`exp LIKE 'masque'`

Filtre la chaîne de caractères ou la date suivant le masque spécifié.

Le masque peut être composé de :

_

Un caractère quelconque.

%

n caractères quelconques, avec n compris entre zéro et l'infini.

[ab...]

Un caractère dans la liste ab... .

[a-z]

Un caractère dans l'intervalle a-z.

[^ab...]

Un caractère en dehors de la liste ou de l'intervalle spécifié.

ab...

Le caractère lui-même.

Pour utiliser _ , %, [et ^ en tant que caractères de recherche, il faut les encadrer de [].

Exemple

```
masque      Chaînes correspondantes
'G%'        commençant par "G"
'_X%1'      deuxième caractère "X", dernier "1"
'%[1-9]'    se terminant par un chiffre compris entre "1" et "9"
'[^XW]%'    ne commençant ni par X ni par W
'LE[_]%'    commençant par "LE_"
```

exp LIKE 'masque' ESCAPE '\'

Lorsque le masque contient un caractère spécial comme _ ou &, il est nécessaire de les faire précéder d'un caractère spécifique appelé caractère d'échappement. Afin de s'adapter à toutes les situations possibles, le caractère d'échappement est libre et il est simplement nécessaire de le spécifier après le mot-clé ESCAPE.

`exp BETWEEN min AND max`

Teste si la valeur de `exp` est incluse dans l'intervalle compris entre les valeurs min et max (bornes incluses).

`EXISTS (sous-requête)`

Renvoie VRAI si la sous-requête a renvoyé au moins une ligne.

Les opérateurs booléens

Ils permettent de combiner des expressions booléennes (expb) en renvoyant une valeur booléenne.

`expb1 OR expb2`

Vrai si une des deux expressions est vraie.

`expb1 AND expb2`

Vrai si les deux expressions sont vraies.

`NOT expb`

Vrai si `expb` est fausse.

1.3 Les fonctions

De nombreuses fonctions sont disponibles pour valoriser des colonnes ou pour effectuer des tests. Des exemples sont donnés avec l'instruction SELECT.

Parmi toutes les fonctions proposées en standard par SQL Server, il est possible de les regrouper par type : rowset, agrégation, ranking, scalaire. Pour ce dernier type de fonctions, il faut en plus les classer par catégories : mathématique, chaîne de caractères, date... car elles sont très nombreuses.

Certaines fonctions, plus particulièrement les fonctions permettant de manipuler les données de type caractère, prennent en compte le classement défini au niveau du serveur.