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");
        }
```

© Editions ENI - All rights reserved

```
        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.

© Editions ENI - All rights reserved

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 :

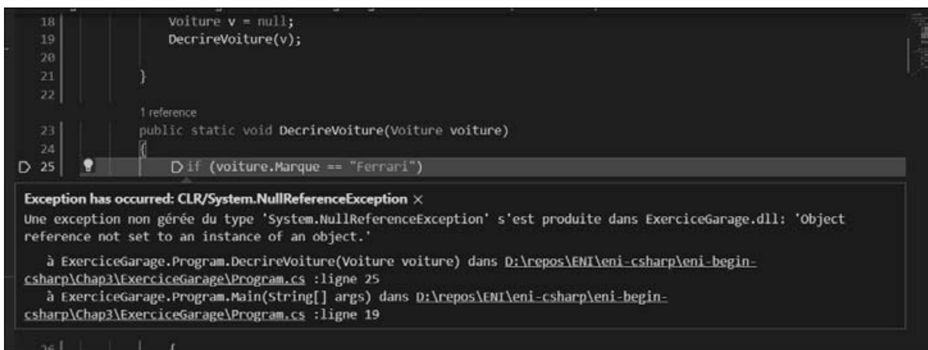
```
test ? cas si vrai : cas si faux
```

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");
    }
}
```

© Editions ENI - All rights reserved

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
L'accès aux données avec ADO.NET

1. Les bases d'ADO.NET

Sous .NET, l'accès aux données s'effectue à l'aide du bloc de services ADO.NET. Bien que le framework ASP.NET ait été enrichi de nouveaux contrôles facilitant la lecture et la présentation des données SQL, le développeur devrait toujours considérer l'emploi du mode connecté pour élaborer une application ASP.NET. En effet, les contraintes de charge, d'intégration et d'exécution du web ont une influence très forte sur l'efficacité finale d'une application ASP.NET.

1.1 Le mode connecté

En mode connecté, tous les formats de base de données adoptent le même fonctionnement. Nous l'illustrerons avec SQL Server, et pour passer à d'autres formats, il n'y aura qu'à changer d'espace de noms et à modifier le préfixe de chaque classe.

Le tableau suivant donne la marche à suivre pour appliquer ces changements :

	Espace de noms	Connexion	Commande
SQL Server	System.Data.SqlClient	**Sql**Connection	**Sql**Command
Access	System.Data.OleDB	**OleDB**Connection	**OleDB**Command
Oracle	System.Data.OracleClient	**Oracle**Connection	**Oracle**Command

Les autres classes sont nommées sur le même principe.

1.1.1 La connexion

La connexion SqlConnection désigne un canal par lequel sont échangés les ordres et les lignes SQL. Ce canal relie le programme C# et la base de données.

L'objet SqlConnection possède plusieurs états, dont deux remarquables : ouvert et fermé. Les autres états sont actifs en régime transitoire ou en cas d'erreur.

Le programme interagit avec une connexion par le biais de la propriété **`ConnectionString`** et des méthodes **`Open()`** et **`Close()`**. L'essentiel des opérations liées à une base ne peut se faire que sur une connexion ouverte.

```
// initialisation de la connexion
string c_string=@"data source=.\SQL2019; initial catalog=
annuaire; integrated security=true";
SqlConnection cx_annuaire;
cx_annuaire=new SqlConnection();
cx_annuaire.ConnectionString=c_string;

// ouverture
cx_annuaire.Open();

// opérations SQL


// Fermeture
cx_annuaire.Close();
```

© Editions ENI - All rights reserved

La chaîne de connexion est formée de différents segments indiquant le nom de la machine abritant la base, le nom de la base, les crédits de l'utilisateur. La syntaxe dépend de chaque format de base. Pour SQL Server, la chaîne de connexion comprend les informations suivantes :

data source	nom de la machine et de l'instance exécutant SQL Server.
initial catalog	nom de la base de données.
integrated security	true ou sspi : la sécurité intégrée est activée. false : la sécurité intégrée n'est pas activée.
user id	nom de l'utilisateur accédant à la base.
password	mot de passe de l'utilisateur accédant à la base.

La documentation MSDN fournit les détails des commutateurs constituant la chaîne de connexion.

Le programmeur se doit d'être particulièrement attentif à la manipulation de la connexion. Une fois ouverte, elle consomme des ressources systèmes et ne peut pas être réouverte avant d'être fermée ; une connexion mal fermée représente ainsi un danger pour l'intégrité et les performances du système.

La syntaxe `try... catch... finally` est alors la seule construction possible pour être certain de fermer une connexion et de libérer des ressources acquises depuis l'ouverture :

```
try
{
    // ouverture
    cx_annuaire.Open();

    // opérations SQL
    // ...
}
catch (Exception err)
{
    Trace.Write(err.Message);
}
finally
{
    try
    {
```

```
            // fermeture
            cx_annuaire.Close();
        }

        catch (Exception err2)
        {
            Trace.Write(err2.Message);
        }
    }
```

Authentification et chaîne de connexion

SQL Server dispose de deux modes d'authentification. Le premier consiste en la fourniture, à chaque connexion, d'un couple (utilisateur, mot de passe) vérifié dans une table des utilisateurs. Cette approche très classique peut s'avérer délicate lorsque les informations de connexion sont consignées dans un fichier de configuration textuel ou lorsqu'elles sont transmises très souvent sur le réseau.

Le deuxième mode d'authentification, appelé sécurité intégrée, n'utilise pas de nom d'utilisateur et de mot de passe transmis sur le réseau. Le système Windows authentifie le programme client (dans notre cas, ASP.NET) et transmet le jeton d'authentification à SQL Server. Ce jeton est codé et d'une durée de vie limitée, ce qui augmente la sécurité d'ensemble.

Lorsque la chaîne de connexion comprend le segment `integrated security=true` (ou `=sspi`), la sécurité intégrée est activée. L'utilisateur ASP.NET doit être préalablement autorisé par SQL Server à l'accès à la base cible. Dans les cas où la sécurité intégrée n'est pas activée, doivent figurer dans la chaîne de connexion les segments `user id=xxx` et `password=xxx`.

```
string c_string = @"data source=.\SQL2019; initial catalog=annuaire;
integrated security=false; user id=sa; password=password";
```

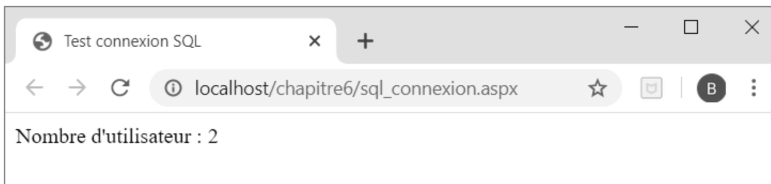
1.1.2 La commande

La commande correspond à une instruction SQL exécutée depuis le programme et appliquée à une base désignée par la connexion associée.

```
// ouverture
cx_annuaire.Open();

// opérations SQL
string rq = "select count(*) from utilisateur";
```

© Editions ENI - All rights reserved

```
SqlCommand sql;
sql = new SqlCommand();
sql.CommandText = rq;
sql.CommandType = CommandType.Text; // valeur par défaut
sql.Connection = cx_annuaire; // association connexion

int cu = (int)sql.ExecuteScalar();
Label1.Text = string.Format("Nombre d'utilisateurs : {0}", cu);

// fermeture
cx_annuaire.Close();
```



La propriété **`CommandType`** précise si **`CommandText`** contient une instruction SQL – comme dans cet exemple – ou bien désigne une procédure stockée. **`CommandType.Text`** étant la valeur par défaut, la ligne d'affectation de CommandType n'est pas nécessaire.

L'objet commande expose quatre méthodes pour exécuter des requêtes SQL. Chacune d'elles est indiquée dans une situation précise :

`ExecuteScalar()`	Exécute une instruction SQL ne retournant qu'une seule valeur (agrégat).
`ExecuteReader()`	Exécute une instruction SQL retournant au moins une valeur ou un enregistrement.
`ExecuteNonQuery()`	Exécute une instruction SQL ne retournant pas de valeur. Convient aux requêtes de création, de mise à jour, aux appels de certaines procédures stockées.
`ExecuteXmlReader()`	Méthode spécifique à SQL Server. Exécute une instruction SELECT et retourne un flux XML.

1.1.3 Le DataReader

Le DataReader – ou plutôt le **`SqlDataReader`** – est un curseur se positionnant d'enregistrement en enregistrement. Il est instancié par la méthode `ExecuteReader`, et avance de ligne en ligne par le biais de la méthode `Read`.

Le programme ne peut qu'avancer le curseur jusqu'à ce qu'il atteigne le dernier enregistrement de la sélection ou qu'il soit fermé. Les données indexées par le curseur ne peuvent pas être modifiées par son intermédiaire. En contrepartie de ces contraintes, le DataReader se révèle être la méthode la plus efficace pour lire des enregistrements.

Exécuter une requête SELECT retournant un DataReader

Les requêtes de type SELECT concernant plusieurs enregistrements ou plusieurs colonnes retournent un DataReader. Leur application suit toujours la même logique :

```
// préparer la connexion
string c_string=@"data source=.\SQL2019; database=annuaire;
integrated security=true";
SqlConnection cx_annuaire=new SqlConnection(c_string);

// une requête select donne lieu à un SqlDataReader
string rq="select idu,nom,telephone,ids from utilisateur";
SqlCommand sql=new SqlCommand(rq,cx_annuaire);

// ouvrir la connexion
cx_annuaire.Open();

// exécuter la requête et récupérer le curseur
SqlDataReader reader=sql.ExecuteReader();

// avancer de ligne en ligne
while(reader.Read())
{

}

// toujours fermer le reader après usage
reader.Close();

// fermer la connexion
cx_annuaire.Close();
```

© Editions ENI - All rights reserved

Comme des clauses `where` ou `having` figurant dans la requête `SELECT` sont à même d'éliminer tous les enregistrements, le DataReader renvoyé est toujours positionné avant le premier enregistrement, s'il existe. La boucle de parcours est donc bien un `while` (et non un `do`).

Il est impératif de fermer le DataReader avant d'exécuter une autre requête sur la même connexion. La structure de contrôle `try... catch ... finally` s'avère alors très utile.

Lire les valeurs des colonnes

Positionné sur un enregistrement, le DataReader se comporte comme un dictionnaire dont les entrées sont indexées par des numéros d'ordre dans la requête (première colonne, seconde colonne...) et par des noms de colonnes. Les syntaxes correspondantes ne sont pas facilement interchangeables et le programmeur doit être très rigoureux quant aux transtypages nécessaires.

```
while(reader.Read())
{
    int idu;
    idu = (int)reader[0];          // syntaxe 1
    idu = (int)reader["idu"];      // syntaxe 2
    idu = reader.GetInt32(0);      // syntaxe 3, sans transtypage
    string nom;
    nom = (string)reader[1];       // syntaxe 1
    nom = (string)reader["nom"];   // syntaxe 2
    nom = reader.GetString(1);     // syntaxe 3
}
```

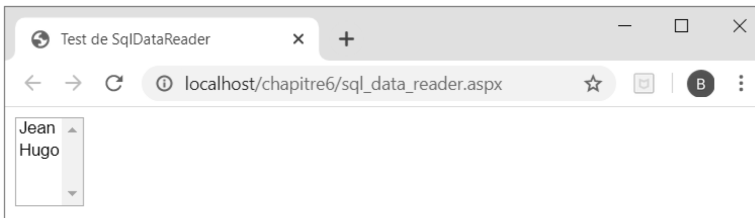
Les trois syntaxes présentées fournissent la même donnée, et globalement avec les mêmes temps de réponse. La deuxième syntaxe, très directe, nécessite quand même un **cast** (transtypage) car l'indexeur de SqlDataReader est typé `object`. Il est très important de distinguer transtypage et conversion. Un transtypage est un mécanisme accordant les types de part et d'autre de l'opérateur d'affectation =. Le compilateur vérifie que le type de la valeur à affecter n'est pas promu sans que le programmeur n'en prenne la responsabilité effective. À l'exécution, le CLR applique une nouvelle vérification et déclenche une exception si le type de la colonne interrogée ne correspond pas au type indiqué dans le programme.

Au contraire, la conversion constitue un changement de type. Pour les types numériques, l'opérateur ( ) est applicable et peut donner lieu à un double transtypage. Pour les types chaînes à convertir en numérique ou en date, les méthodes d'analyse textuelle (parsing) sont exposées par les types cibles (`int.Parse` ou `DateTime.Parse`).

La troisième syntaxe ne donne pas lieu à un transtypage car le DataReader expose des méthodes spécifiques à chaque type. Néanmoins, le framework vérifiera lors de l'exécution que ces méthodes ont été appliquées à bon escient. Il n'est pas possible de lire un nombre double à partir d'une colonne **`varchar`** sans opérer de conversion.

Finalement, le programmeur applique la syntaxe qui lui paraît la plus directe, ceci n'a pas beaucoup de conséquences pour le reste du code.

```
while(reader.Read())
{
    int idu;
    idu = (int)reader["idu"];

    string nom;
    nom = (string)reader["nom"];

    ListBox1.Items.Add(new ListItem( nom, idu.ToString()));
}
```



© Editions ENI - All rights reserved