Chapitre 3
Programmation orientée objet

1. Principes de la programmation orientée objet

La programmation orientée objet (POO) est un paradigme très répandu en développement logiciel. Il vient compléter un panorama déjà riche du paradigme procédural ainsi que du paradigme fonctionnel.

La POO est une forme de conception de code visant à représenter les données et les actions comme faisant partie de classes, elles-mêmes devenant des objets lors de leur création en mémoire. Cette notion a été rapidement présentée dans le chapitre précédent, il est maintenant temps de comprendre son fonctionnement plus en détail.

1.1 Qu'est-ce qu'une classe ?

Une classe est un élément du système que forme votre application. Une classe contient deux types d'élément de code : des données ainsi que des méthodes, représentant des actions. Il faut voir la classe comme étant une boîte dans laquelle il est possible de ranger ces deux types d'éléments. Pour faire un parallèle avec la vie réelle, nous pouvons facilement comprendre que la définition d'une classe s'applique à un objet comme un ordinateur, par exemple. Ce dernier dispose de méthodes (allumer, éteindre...) ainsi que des propriétés (nombre d'écrans, quantité de RAM...).

Conceptuellement, une classe n'est qu'une définition. Une fois que vous avez statué sur ce qu'elle doit contenir ainsi que ses méthodes, il convient de la créer. Cette action s'appelle l'instanciation. À la suite de cette opération, nous obtenons une instance en mémoire d'un objet.

Pour tenter une comparaison, prenons l'exemple d'une usine de fabrication d'objets en bois. Afin de pouvoir créer un objet, il faut un plan (la classe). Grâce à ce dernier, la machine peut découper et assembler les divers éléments (les données et méthodes) afin de créer une nouvelle instance (instanciation).

En C#, la déclaration d'une classe se fait grâce au mot-clé `class`. Il y a quelques spécificités possibles, notamment la portée, que nous étudierons juste après, dans la section Que peut-on déclarer dans une classe ? - Les méthodes, ainsi que les concepts de `static`, `sealed` et celui de `partial`. La syntaxe complète de la déclaration d'une classe est la suivante :

```
PORTÉE [static] [sealed] [partial] class NOM_CLASSE
```

Le nom de la classe est libre mais répond à deux règles :

– Il ne peut contenir que des caractères alphanumériques et le signe underscore (« _ »).

– Il ne peut pas commencer par un chiffre.

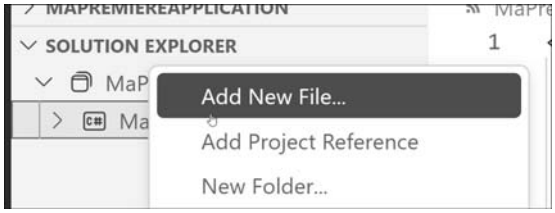
En plus de ces règles, les développeurs C# respectent souvent une convention syntaxique : l'utilisation du PascalCase. Cela indique que le nom commence par une majuscule et chaque mot est symbolisé par une majuscule également, par exemple, `OrdinateurPortable`. Le langage et le compilateur n'interdisent pas d'écrire `ordinateurPortable`, `Ordinateurportable` ou encore `ordinateurportable`, mais ces différentes déclarations ne respectent pas la convention largement admise et appliquée. Finalement, bien que ce soit possible, il est recommandé d'éviter tout caractère accentué dans le nom d'une classe. Par exemple, il est préférable d'appeler sa classe `Pieton` plutôt que `Piéton`, cela afin que le code C# produit soit le plus proche possible de ce que nous aurions en anglais.

Dans le programme de base créé en C# dans le précédent chapitre, une classe `Program` est créée par défaut. Nous pouvons constater qu'il n'y a ni notion de portée ni notion de `partial` ou `static`. Une fois qu'une classe est déclarée, elle définit un bloc, dans lequel nous pouvons implémenter les données et les méthodes dont notre programme a besoin pour fonctionner.

1.1.1 Les classes dans Visual Studio Code

Pour créer une classe dans Visual Studio Code, il est nécessaire de suivre les étapes suivantes :

- Dépliez la vue **Solution Explorer** du projet.
- Placez-vous sur le dossier où vous souhaitez créer la nouvelle classe (ou directement sur le nom du projet si vous souhaitez la créer à la racine).
- Faites un clic droit pour sélectionner l'élément de menu **Add New File**.
- Renseignez le nom de la classe dans la petite pop-up qui s'est ouverte en haut au centre de l'écran (toujours sans espaces ni caractères spéciaux).



Ajout d'une nouvelle classe avec Visual Studio Code

À la suite de ces manipulations, un nouveau fichier, portant le nom de la classe suivi de l'extension .cs, est disponible dans la hiérarchie à gauche. Par défaut, ce fichier sera ouvert et contient la classe qui a été déclarée dans l'espace de noms correspondant au dossier de destination.

1.1.2 L'héritage

Il existe un concept extrêmement important en POO : l'héritage. Globalement, si vous avez la possibilité de dire « X est un Y », l'équivalent pourrait être de dire « X hérite de Y ». X reprend toutes les propriétés et tous les comportements de Y, mais le spécifie. Donnons un exemple concret : « Un Mac est un ordinateur ». Donc, au niveau du développement orienté objet, un Mac reprend toutes les propriétés d'un ordinateur ainsi que ses comportements, mais les spécifie en y apportant ses propres éléments. On dit dans ces cas-là que `Mac` est une classe fille de la classe `Ordinateur`.

En C#, cette notion est centrale car tous les éléments que vous allez manipuler héritent naturellement de la classe `System.Object`, qui définit le comportement de base de n'importe quel objet. De surcroît, contrairement à d'autres langages (comme le C++), il n'est pas possible, en C#, d'hériter de plusieurs classes : une seule classe mère est possible. Si aucune classe mère n'est spécifiée, c'est par définition la classe `System.Object` qui constitue la classe mère (sans qu'une quelconque manipulation soit requise).

Remarque

En C#, il n'est pas possible d'hériter de plusieurs classes. Il faut donc choisir la classe dont on hérite. En l'absence de précision, le compilateur génère automatiquement, de façon transparente, un héritage de la classe `System.Object`, comme décrit ci-dessus. Si nous spécifions un héritage, cela ne veut pas dire que la classe hérite de `System.Object` ET de la classe héritée, mais uniquement de la classe héritée, qui remplace l'héritage généré par le compilateur. La classe héritée, elle-même, hérite soit d'une autre classe, soit directement de `System.Object`. En finalité, toutes les classes en C# héritent d'une façon ou d'une autre de `System.Object`.

Afin d'indiquer qu'une classe hérite d'une autre, il faut utiliser le deux-points, suivi de la classe dont on souhaite hériter :

```
class Ordinateur { }
class Mac : Ordinateur { }
```

Bien entendu, ce n'est pas parce qu'une classe hérite d'une autre qu'elle a forcément accès à tout ce qui a été défini au sein de la classe mère.

1.1.3 L'encapsulation

Tout ce qui se trouve à l'intérieur d'une classe est désigné par un terme bien spécifique : l'encapsulation. Avec celle-ci vient également la notion de portée, qui indique comment les choses sont perçues d'un point de vue extérieur à la classe.

La portée permet de définir la visibilité d'un élément d'une classe ou de la classe elle-même. Il existe en tout sept portées en C# :

- `public` : définit que l'élément est totalement visible dans et en dehors de la classe.
- `private` : définit que l'élément n'est visible qu'au sein de la classe où il est déclaré alors qu'il est totalement invisible de l'extérieur.
- `internal` : définit que l'élément est visible uniquement au sein du projet où il est déclaré. Nous pouvons considérer l'élément comme étant public, mais simplement au sein du projet dans lequel il est déclaré. Un autre projet qui référence notre projet n'a pas connaissance d'un élément déclaré comme `internal`. Par défaut, en l'absence de portée explicite sur une classe, c'est la portée `internal` qui est sélectionnée par le compilateur.
- `protected` : définit que l'élément est visible uniquement au sein de la classe où il est déclaré ainsi que dans sa hiérarchie de classes filles. Cela rejoint le concept de l'héritage, que nous verrons plus loin dans ce chapitre.
- `protected internal` : définit un cumul entre `protected` et `internal`. Un élément déclaré avec cette portée est visible par la classe concernée ainsi que ses classes filles, tout comme par toutes les autres classes au sein du même projet. Cela signifie également que si une classe fille est déclarée en dehors du projet actuel, elle peut accéder à un élément `protected internal`, tout comme n'importe quelle classe du même projet.
- `private protected` : définit une intersection entre `protected` et `internal`. Un élément déclaré avec cette portée n'est visible que par la classe concernée ainsi que les classes filles qui sont définies au sein du même projet. Cela veut dire qu'une classe fille définie en dehors du projet actuel ne pourra pas accéder à cet élément.

– `file` : ajoutée en C# 11, cette portée définit une visibilité uniquement dans le cadre du fichier en cours. Cette portée est très particulière car elle n'a pas vocation à être directement utilisée par les développeurs. Elle existe surtout pour les outils de génération automatique de code. Néanmoins, dans de très rares cas, il peut être utile de déclarer un élément qui n'existe que dans le cadre d'un fichier pour un algorithme précis. Il est à noter également que, contrairement aux autres portées, cette portée n'est valide que pour la déclaration d'un type. Elle ne peut pas s'appliquer sur une méthode, un champ ou une propriété.

Avec toutes ces portées, il est possible de créer la classe qui correspond finement au besoin de votre application, pour éviter que certains éléments ne sortent du périmètre de la classe. En reprenant notre exemple, considérons que la classe `Ordinateur` dispose d'un booléen indiquant si la machine est allumée ou non. Afin d'éviter que quelqu'un ne puisse manipuler directement cette donnée, la manière de procéder est de la définir comme étant publiquement accessible en lecture, mais privée pour ce qui est de l'écriture. En conséquence, seule une méthode publique, définie dans cette classe, comme par exemple `Allumer` ou `Eteindre`, peut changer la valeur de cet indicateur. Nous nous préservons ainsi d'un changement d'état non maîtrisé (car nous pouvons considérer que l'opération d'extinction nécessite d'effectuer quelques opérations en amont avant de basculer le booléen).

1.2 Que peut-on déclarer dans une classe ?

Nous l'avons vu, il existe deux types d'éléments que nous pouvons déclarer dans une classe : des méthodes (actions) et des données. Voyons rapidement comment les déclarer.

1.2.1 Les méthodes

Une méthode traduit une action qu'il est possible d'invoquer sur la classe. Lors de la déclaration d'une méthode, il faut se poser les questions suivantes :

– S'agit-il d'une action qui doit pouvoir être réalisée depuis l'extérieur ou uniquement depuis l'intérieur de la classe ?

– Est-ce qu'une valeur de retour particulière est attendue ?

– Certaines informations sont-elles nécessaires pour que cette méthode fonctionne ?

Vous avez déjà eu un aperçu d'un appel de méthode dans le premier chapitre, sur la classe `Console` : `WriteLine` et `ReadLine`. Ces deux méthodes illustrent bien les points cités précédemment :

– `WriteLine` doit pouvoir être appelée depuis l'extérieur. Nous n'attendons pas de valeur en retour à son appel, mais il est nécessaire de lui transmettre l'information que nous souhaitons écrire.
– `ReadLine` doit également pouvoir être appelée depuis l'extérieur. Nous avons besoin de récupérer l'information saisie par l'utilisateur uniquement, sans besoin de lui transmettre une quelconque information.

La syntaxe de déclaration d'une méthode dans une classe est la suivante :

```
PORTÉE [static] TYPE_RETOUR NOM_METHODE([PARAMÈTRES])
```

Le type de retour doit correspondre à un type C# connu. Par exemple, si nous souhaitons créer une méthode qui réalise l'addition de deux nombres et renvoie le résultat, le tout accessible publiquement, nous la déclarons comme suit :

```
public int Addition(int premier, int second) {}
```

Remarque

Dès lors que nous déclarons une méthode avec une valeur de retour sans écrire le contenu de la méthode, le compilateur émet immédiatement une erreur de compilation. Ceci est dû au fait que chaque méthode retournant un résultat doit obligatoirement comporter une instruction `return`.

Lorsqu'une méthode doit renvoyer une valeur, il faut utiliser le mot-clé `return` afin de définir la valeur que nous souhaitons renvoyer. L'instruction `return` peut être utilisée directement avec une valeur ou alors nous pouvons nous servir d'une variable du type de retour attendu. Dans le cas de l'exemple ci-dessus, ces deux façons d'écrire la méthode sont valides :

```
public int Addition(int premier, int second)
{
    return premier + second;
}
```

```
public int Addition(int premier, int second)
{
    int resultat = premier + second;
    return resultat;
}
```

Un élément important à garder en mémoire : à l'instar de ce que nous avons vu dans le chapitre précédent avec la déclaration de classes du même nom au sein du même espace de noms, il n'est pas possible de déclarer deux fois la même méthode à l'intérieur d'une même classe. Si les noms sont identiques et que les paramètres le sont également, alors le compilateur C# considère qu'il s'agit de la même méthode. La valeur de retour ne constitue pas un élément distinctif. Ainsi, la déclaration des deux méthodes suivantes dans la même classe est impossible et cela provoque une erreur de compilation :

```
public int Addition (int premier, int second)
{
    return premier + second;
}
public void Addition (int premier, int second)
{
}
```

Remarque

Comme nous pouvons le constater dans l'exemple ci-dessus, le mot-clé void précise que la méthode ne renvoie aucun résultat. La notion de type de retour étant obligatoire, il faut utiliser ce mot-clé pour indiquer les cas où il n'y en a pas.

Si la méthode ne prend pas de paramètres, la présence de parenthèses ouvrantes et fermantes accolées au nom de la méthode est malgré tout nécessaire pour signifier qu'il s'agit d'une méthode :

```
public void MaMethode()
{
}
```

Au sein d'une méthode qui déclare son propre bloc, il est possible de déclarer des variables et constantes qui sont considérées uniquement comme locales (c'est-à-dire visibles au sein de la méthode et de tous ses sous-blocs, mais invisibles dans les blocs parents, directs ou indirects).

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();
```

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";
```

```
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();
```

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()));
}
```