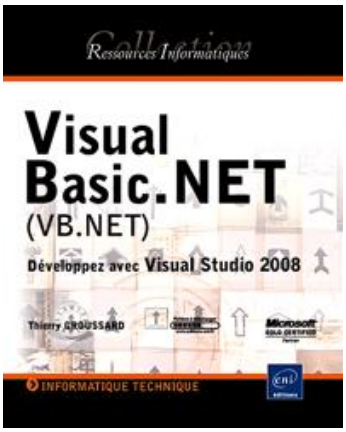
Visual Basic.NET

(VB.NET)

Thierry GROUSSARD



Résumé

Ce livre sur VB.Net s'adresse aux développeurs, même débutants, désireux de maîtriser Visual Basic.NET. Après une description de l'environnement de développement (**Visual Studio 2008**), le lecteur découvrira les bases de la **programmation orientée objet** avec VB.NET. Il évoluera de façon progressive vers sa mise en œuvre avec le développement d'applications **Windows Form**. Les nombreux exemples et les conseils sur l'utilisation des outils de débogage lui fourniront une aide précieuse pendant la réalisation d'une application.
Un chapitre consacré à l'accès aux bases de données à l'aide de **ADO.NET 2.0 et de SQL** permettra d'évoluer vers le développement d'applications client-serveur. Les **puissantes fonctionnalités de LINQ** sont présentées et détaillées pour faciliter l'accès et la manipulation des données. Le **langage XML** est également présenté permettant ainsi de faciliter l'échange d'informations avec d'autres applications.
Les utilisateurs des versions précédentes découvriront les nouveautés et améliorations de cette version 2008 (**types nullables, méthodes partielles, classes anonymes, ...**) leur permettant de développer encore plus rapidement et facilement des applications pour le **framework .NET 3.5** et pour **Windows Vista**.
La distribution d'une application est présentée avec l'utilisation de **Windows Installer** et de la technologie **Click Once**.
Les exemples cités dans le livre sont en téléchargement sur cette page.

L'auteur

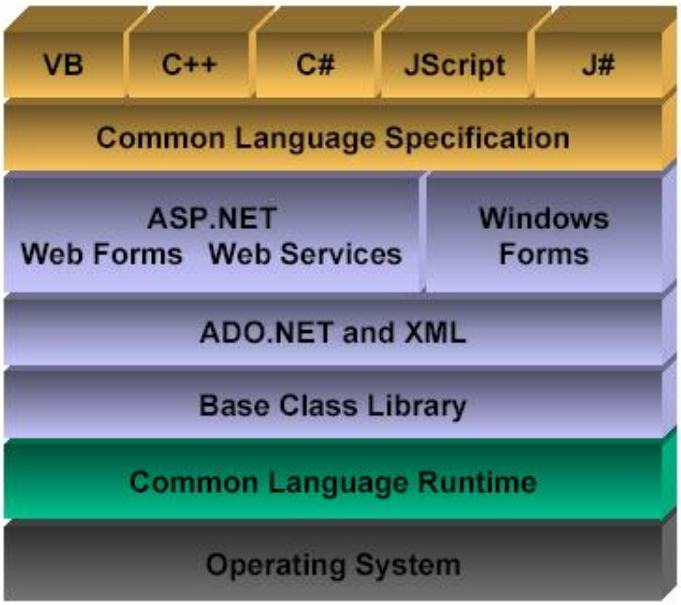
Analyste et développeur pendant plus de 10 ans, **Thierry Groussard** s'est ensuite orienté vers la formation, et plus particulièrement dans le domaine du développement. Sa connaissance approfondie des besoins de l'entreprise et ses qualités pédagogiques rendent cet ouvrage particulièrement adapté à l'apprentissage et à la mise en pratique du développement sous VB.NET 2008.

Introduction

La plate-forme .NET fournit un ensemble de technologies et d'outils facilitant le développement d'applications et propose une solution pour pratiquement tous les types d'applications :

- applications Windows classiques ;
- application Web ;
- services Windows ;
- services Web.

Tous ces types d'applications sont réalisables grâce à un élément essentiel : le Framework .NET. Ce Framework prend en charge, par l'intermédiaire de nombreuses couches logicielles superposées, l'intégralité de la vie d'une application, du développement jusqu'à l'exécution. Le framework doit être hébergé par un système d'exploitation avec lequel il va interagir. Le premier système permettant de l'accueillir est bien sûr Windows mais d'autres versions sont disponibles permettant l'adaptation de la plate-forme .NET à des systèmes tels Linux ou Unix.



Le framework contient deux éléments principaux : le Common Language Runtime et la bibliothèque de classes du .NET Framework.

Le Common Language Runtime est la base du .NET Framework. Le runtime peut être considéré comme un moteur d'exécution qui gère l'exécution du code et assure également la gestion de la mémoire. Le code pris en charge par le Common language Runtime est appelé code managé.

La bibliothèque de classes est un ensemble de classes pouvant être utilisé pour le développement de tout type d'application. Nous le manipulerons tout au long de cet ouvrage.

1. Principe de fonctionnement du Common Language Runtime

Dans les applications Windows traditionnelles, le système prend directement en charge l'exécution du code. En effet, celui-ci est généré par le compilateur associé au langage de programmation utilisé pour la conception de l'application. Le résultat de cette compilation correspond à un fichier binaire contenant le code spécifique pour le microprocesseur et le système d'exploitation avec lesquels l'application doit fonctionner. Aucune compatibilité avec un autre type de microprocesseur ou système d'exploitation n'est possible. Pour s'en convaincre, il suffit de tenter l'exécution d'une application prévue pour Windows sur un système Linux pour vérifier cette incompatibilité. Si l'on tente l'exécution sur une station de travail SUN, qui utilise un type de microprocesseur radicalement différent, le résultat est identique. La solution pour s'affranchir de ces problèmes consiste à générer à la compilation, non pas un code spécifique, mais un code générique, indépendant de toute plate-forme logicielle ou matérielle. Ce code est, au moment de l'exécution, confié à une machine virtuelle qui en assure l'exécution. Ce code s'appelle Microsoft Intermediate Language (MSIL). Lors de l'exécution de l'application, ce code est pris en charge par la machine virtuelle qui en assure la traduction en

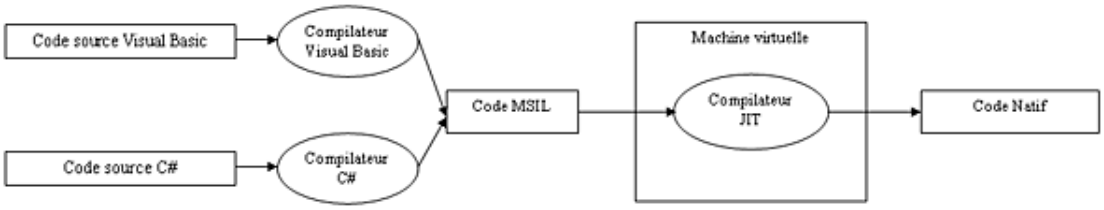
instructions spécifiques pour le microprocesseur de la machine. Cette traduction n'est pas effectuée, en bloc dès le début de l'application, mais uniquement au fur et à mesure des besoins. En effet, pourquoi perdre du temps à traduire du code MSIL, s'il n'est jamais utilisé par la suite. C'est pour cette raison que le compilateur utilisé pour cette traduction s'appelle compilateur Just In Time (JIT).

Les avantages de cette solution sont évidents car pour exécuter une même application sur plusieurs plates-formes matérielles et ou logicielles, il suffit d'obtenir la machine virtuelle capable d'effectuer la traduction. Cette machine virtuelle est disponible pour tous les systèmes Microsoft. Le projet Mono propose une version de la machine virtuelle pour les plates-formes suivantes :

- Linux
- Mac OS X
- Sun Solaris
- BSD - OpenBSD, FreeBSD, NetBSD

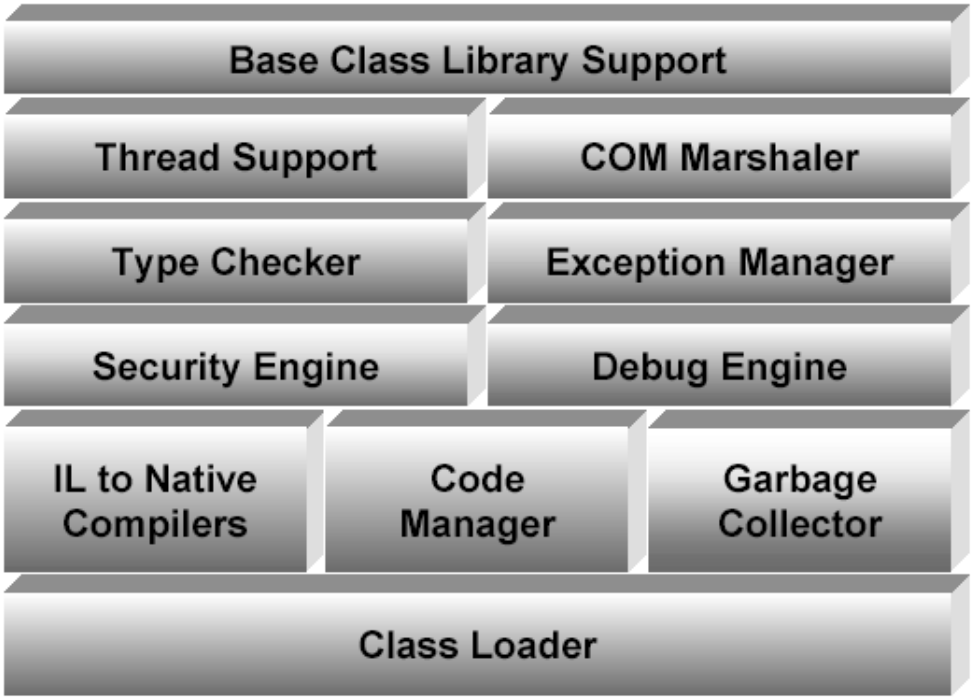
Elles sont disponibles en téléchargement sur le site http://www.mono-project.com

Le schéma suivant reprend l'ensemble de ces opérations :



2. Les services du Common Language Runtime

La machine virtuelle ne se contente pas d'effectuer la traduction du code. Le code MSIL est aussi appelé code managé, ce qui sous-entend qu'un certain nombre d'opérations supplémentaires seront réalisées sur le code au moment de l'exécution. La figure ci-après reprend l'ensemble des fonctionnalités disponibles dans le Common Language Runtime.



Class Loader

Il gère le chargement en mémoire des instances de classes.

IL To Native Compilers

Il convertit le code intermédiaire (MSIL) en code natif.

Code Manager

Il gère l'exécution du code.

Garbage Collector

Il assure la gestion de la mémoire en surveillant les instances de classes qui ne sont plus accessibles.

Security Engine

Il permet la vérification de l'identité de la personne demandant l'exécution du code et accepte ou non cette exécution, en fonction des autorisations accordées.

Debug Engine

Il permet le débogage de l'application, en prenant en charge par exemple l'exécution en pas à pas du code.

Type Checker

Il surveille l'utilisation de variables non initialisées et les conversions entre des variables de type différent.

Exception Manager

Il fournit la gestion structurée des exceptions en liaison avec Windows Structured Exception Handling (SEH). Cette technique permet une gestion individuelle de chaque exception plutôt qu'une gestion globale.

Thread Support

Il propose un ensemble de classes permettant la réalisation d'applications multithread.

COM Marshaler

Il permet de traduire des appels vers des composants COM, assurant par exemple la conversion des types de données.

Base Class Library Support

Il fournit l'accès aux services disponibles sur le système d'exploitation hôte.

3. La Base Class Library

Le Framework .NET met à la disposition des développeurs un ensemble d'outils lui permettant d'obtenir une solution rapide à une majorité de problèmes rencontrés lors de la réalisation d'une application.

Ces outils sont disponibles sous forme de classes. À l'inverse des bibliothèques de code des langages de la génération précédente, qui n'étaient qu'une liste interminable de procédures ou fonctions, la bibliothèque de classes est organisée sous forme d'une structure hiérarchisée. L'élément essentiel de cette hiérarchisation est l'espace de nom (Namespace). Il permet le regroupement logique de classes ayant des points communs. Par exemple, on retrouve dans le namespace System.Data toutes les classes utilisables pour accéder à une base de données.

Cette bibliothèque de classes est bien sûr indépendante d'un quelconque langage de programmation. Elle permet donc le mélange de différents langages au cours du développement d'une application. Elle est également parfaitement intégrée à Visual Studio, ce qui nous procure un confort d'utilisation appréciable avec des outils comme Intelissence. Comme cette librairie est orientée objet, elle est facilement extensible par le biais de relations d'héritage.

La bibliothèque contient une quantité impressionnante d'espaces de nom et de classe, tant et si bien que, au cours de vos développements avec Visual Basic, il y a de fortes chances pour que vous n'utilisiez jamais certains d'entre eux.

Les espaces de noms les plus utilisés sont les suivants :

System

C'est l'espace de nom racine pour les types de données dans le Framework .NET. Il contient notamment la définition de la classe Object, qui est l'ancêtre de tous les types de données du Framework .NET.

System.Windows

Il contient l'ensemble des éléments permettant la création d'interfaces utilisateurs Windows.

System.Web

Il contient toutes les ressources nécessaires pour la création d'applications Web, avec par exemple, les classes de la technologie ASP.NET ou les classes utilisables pour la création de services Web XML.

System.data

Il contient un ensemble de classes spécialisées dans l'accès aux bases de données, avec le support de ADO.NET.

System.Xml

Le langage Xml est devenu omniprésent et cet espace de nom contient les classes assurant la manipulation de documents Xml.

4. Les versions et évolutions de la plate-forme .NET

La première version (1.0) de la plate-forme .NET sort en janvier 2002 avec Visual Studio 2002. Cette version est rapidement remplacée par la version 1.1 qui corrige quelques petits problèmes de jeunesse de la version précédente et ajoute des technologies qui n'étaient auparavant disponibles qu'en tant qu'installations indépendantes et sont désormais incluses. Les apports de cette version sont principalement :

- Les Contrôles mobiles ASP.NET (anciennement *Microsoft Mobile Internet Toolkit*) qui étendent le Framework .NET par la prise en charge des périphériques mobiles (sans fil) tels que téléphones portables et assistants numériques personnels.
- Le fournisseur de données .NET Framework pour ODBC et le fournisseur de données pour Oracle qui auparavant n'étaient disponibles que par téléchargement, sont désormais livrés avec le .NET Framework.
- La prise en charge de la nouvelle mise à jour du protocole Internet couramment appelée IP version 6 ou plus simplement IPv6. Ce protocole est conçu pour augmenter sensiblement l'espace d'adressage qui est utilisé pour identifier les points d'entrée de communication d'Internet.

Elle est disponible avec la version 2003 de Visual Studio en avril 2003.

Il faut attendre novembre 2005 pour voir arriver la version 2.0 associée à la sortie de Visual Studio 2005. Cette version apporte de nombreuses améliorations :

- La prise en charge de la nouvelle génération d'ordinateurs 64 bits permettant la création d'applications plus performantes.
- Une évolution majeure dans l'accès aux bases de données avec ADO.NET 2.0 améliorant l'utilisation de XML.
- Le développement d'applications Web est également de plus en plus facile avec la nouvelle version de ASP.NET proposant une multitude de nouveaux contrôles.
- L'utilisation de la classe Console est optimisée avec l'ajout de nouvelles propriétés et méthodes (gestion des couleurs, effacement, position du curseur…).
- Le .NET Framework 2.0 réintroduit la fonctionnalité Modifier & Continuer permettant à l'utilisateur qui débogue une application dans Visual Studio de modifier le code source en mode arrêt. Une fois les modifications du code source appliquées, l'utilisateur peut reprendre l'exécution du code et observer l'effet.
- L'apparition de la notion de générique qui permet aux classes, structures, interfaces, méthodes et délégués d'être déclarés et définis avec des paramètres de type non spécifié ou générique au lieu de types spécifiques. Les types réels sont spécifiés ultérieurement lors de l'utilisation.

La version 3.0 arrive en novembre 2006 et apporte de nouvelles technologies tout en restant à la base une version 2.0. Ces technologies sont disponibles sous forme de téléchargements qui viennent s'intégrer au framework 2.0. Voici un bref aperçu de ces nouveautés :

- *Windows Presentation Foundation* (WPF) représente le nouveau système d'interfaces graphiques. Il se base sur un moteur de rendu vectoriel et permet une séparation plus claire entre la définition de l'interface graphique d'une application et son code. Il utilise pour cela le langage XAML (*eXtensible Application Markup Language*). Les tâches peuvent ainsi être plus facilement réparties entre designers et développeurs.

- *Windows Communication Foundation* (WCF) constitue la nouvelle base de développement d'applications distribuées. Il facilite la communication entre applications en ajoutant une couche d'abstraction uniformisant les techniques de communication entre applications (Services Web, .NET Remoting, Microsoft Transaction Server, et Microsoft Message Queuing,…).

- *Windows Workflow Foundation* (WF) est composé d'un modèle de programmation, d'un moteur d'exécution et d'outils pour intégrer des workflows dans une application. Un workflow peut être défini comme un ensemble d'actions ou étapes s'exécutant dans un ordre prédéfini. Ces actions peuvent s'enchaîner en fonction de conditions, d'interactions avec des processus informatiques ou en fonction d'interactions humaines.

- Windows Cardspace fournit une nouvelle technique aux utilisateurs pour s'identifier dans une application. Elle a la même vocation que Microsoft Passport mais n'est pas spécifique aux applications Microsoft (Hotmail, MSDN…).

Écriture, compilation et exécution d'une application

Dans ce chapitre nous allons détailler le cycle de vie d'une application, depuis la rédaction du code jusqu'à l'exécution de l'application, en étudiant en détail les mécanismes mis en œuvre.

1. Écriture du code

L'immense majorité des applications sont développées grâce à un environnement intégré qui regroupe les principaux outils nécessaires, à savoir :

- un éditeur de texte ;
- un compilateur ;
- un débogueur.

Cette approche est de loin la plus confortable. Elle nécessite cependant une petite phase d'apprentissage pour se familiariser avec l'outil. Pour notre première application, nous allons utiliser une démarche un petit peu différente puisque nous allons utiliser des outils individuels : le bloc-notes de Windows pour l'écriture du code et le compilateur en ligne de commandes pour Visual Basic.

Notre première application sera très simple puisqu'elle affichera simplement le message "Bonjour" dans une fenêtre de commande.

Voici le code de notre première application que nous allons ensuite expliquer ligne par ligne. Il est à saisir à l'aide du bloc-notes de Windows ou de tout autre éditeur de texte, à condition que celui-ci ne rajoute pas de code de mise en page à l'intérieur du document, comme le font par exemple les logiciels de traitement de texte.

Exemple

```
Imports System
  public Module test
    dim message as string="bonjour"
    public sub main ()
      console.writeline(message)
    end sub
end module
```

Ce code est à sauvegarder dans un fichier portant l'extension .vb. Cette extension n'est pas obligatoire, mais elle permet de respecter les conventions utilisées par Visual Studio. Détaillons maintenant les quelques lignes de notre première application.

```
Imports System
```

Cette ligne permet de rendre directement accessibles les éléments présents dans le namespace System. Sans elle, il faudrait utiliser les noms pleinement qualifiés pour tous les éléments contenus dans le namespace. Dans notre cas, nous devrions alors utiliser : `System.Console.writeline("Bonjour")`

```
public Module test ... end module
```

Dans Visual Basic, toute portion de code doit être contenue dans un module ou une classe.

```
dim message as string="bonjour"
```

Cette ligne déclare une variable. Toutes les variables doivent être déclarées avant de pouvoir être utilisées. La déclaration permet de spécifier le type d'information que la variable va contenir, ici une chaîne de caractères, et éventuellement une valeur initiale, "bonjour" dans notre cas.

```
public sub Main() ... end sub
```

Toutes les instructions autres que des déclarations doivent être placées dans une procédure ou une fonction. La majeure partie du code est donc placée entre les instructions Sub et End Sub ou Function et End Function. Parmi toutes ces procédures et fonctions, l'une d'entre elles est désignée comme le point d'entrée dans l'application. C'est par

l'exécution de cette procédure que démarre l'application. Cette procédure doit être publique et doit s'appeler Main.

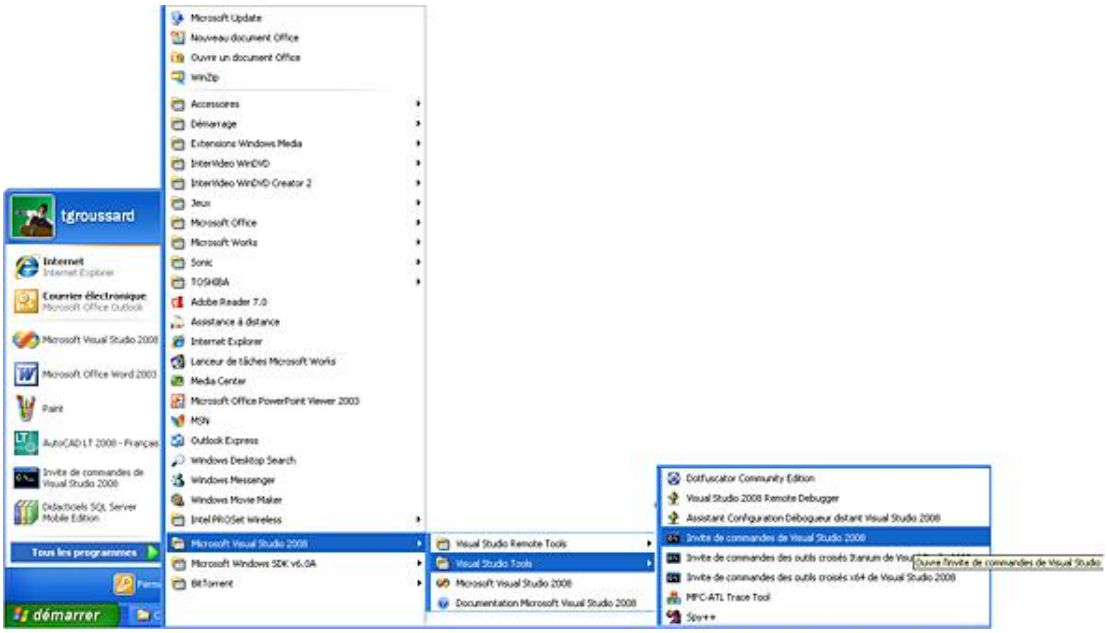
```
Console.writeline("Bonjour")
```

La classe Console définie dans l'espace de nom System fournit un ensemble de méthodes permettant l'affichage d'informations sur la console ou la lecture d'informations depuis la console. La procédure writeline permet l'affichage d'une chaîne de caractères sur la console.

À noter également que Visual Basic ne fait pas de distinction entre les minuscules et les majuscules dans les instructions. Si vous utilisez l'éditeur de Visual Studio pour rédiger votre code, celui-ci fera automatiquement les modifications pour uniformiser "l'orthographe" de votre code.

2. Compilation du code

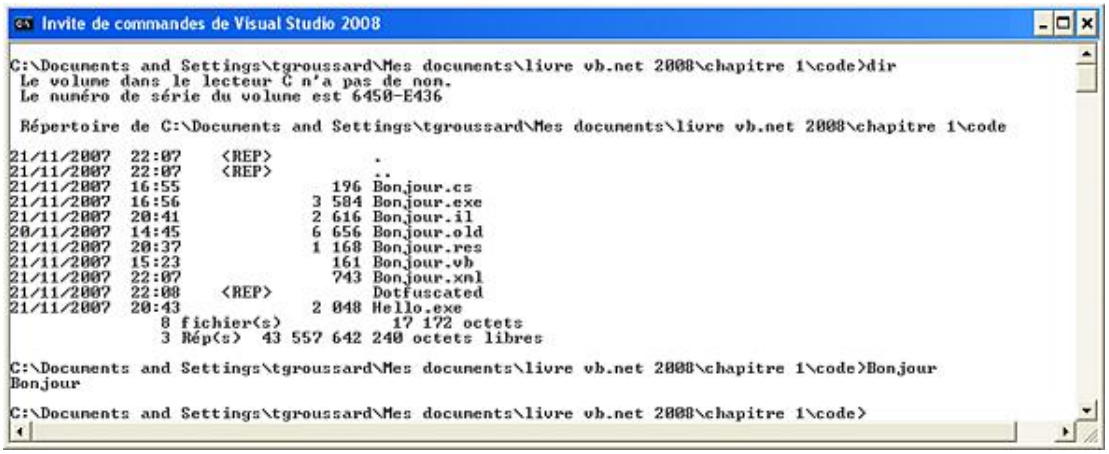
Le Framework .NET inclut un compilateur en ligne de commande pour Visual Basic. Pour compiler le code source de notre exemple, nous devons ouvrir une fenêtre de commande DOS pour pouvoir lancer le compilateur. Pour cela, un raccourci a été créé dans le menu démarrer pendant l'installation. Ce raccourci lance l'exécution d'un fichier .bat qui positionne certaines variables d'environnement nécessaires pour le bon fonctionnement des outils Visual Studio en ligne de commande.



À partir de la fenêtre de commande ouverte, il convient de se placer dans le répertoire dans lequel se trouve le fichier source.

La compilation est lancée par la commande `vbc Bonjour.vb`.

Après un bref instant, le compilateur nous rend la main. Nous pouvons vérifier la présence du fichier exécutable et vérifier son bon fonctionnement.



Notre première application est vraiment très simple. Pour des applications plus évoluées, il sera parfois utile de spécifier certaines options pour le fonctionnement du compilateur. L'ensemble des options disponibles peut être obtenu en lançant la commande `vbc / ?`.

Les principales options sont :

`/out:fichier.exe`

Cette option permet de spécifier le nom du fichier résultat de la compilation. Par défaut, c'est le nom du fichier source en cours de compilation qui est utilisé.

`/target :exe`

Cette option demande au compilateur la génération d'un fichier exécutable pour une application en mode console.

`/target :winexe`

Cette option demande au compilateur la génération d'un fichier exécutable d'application Windows.

`/target :library`

Cette option demande au compilateur la génération d'un fichier bibliothèque dll.

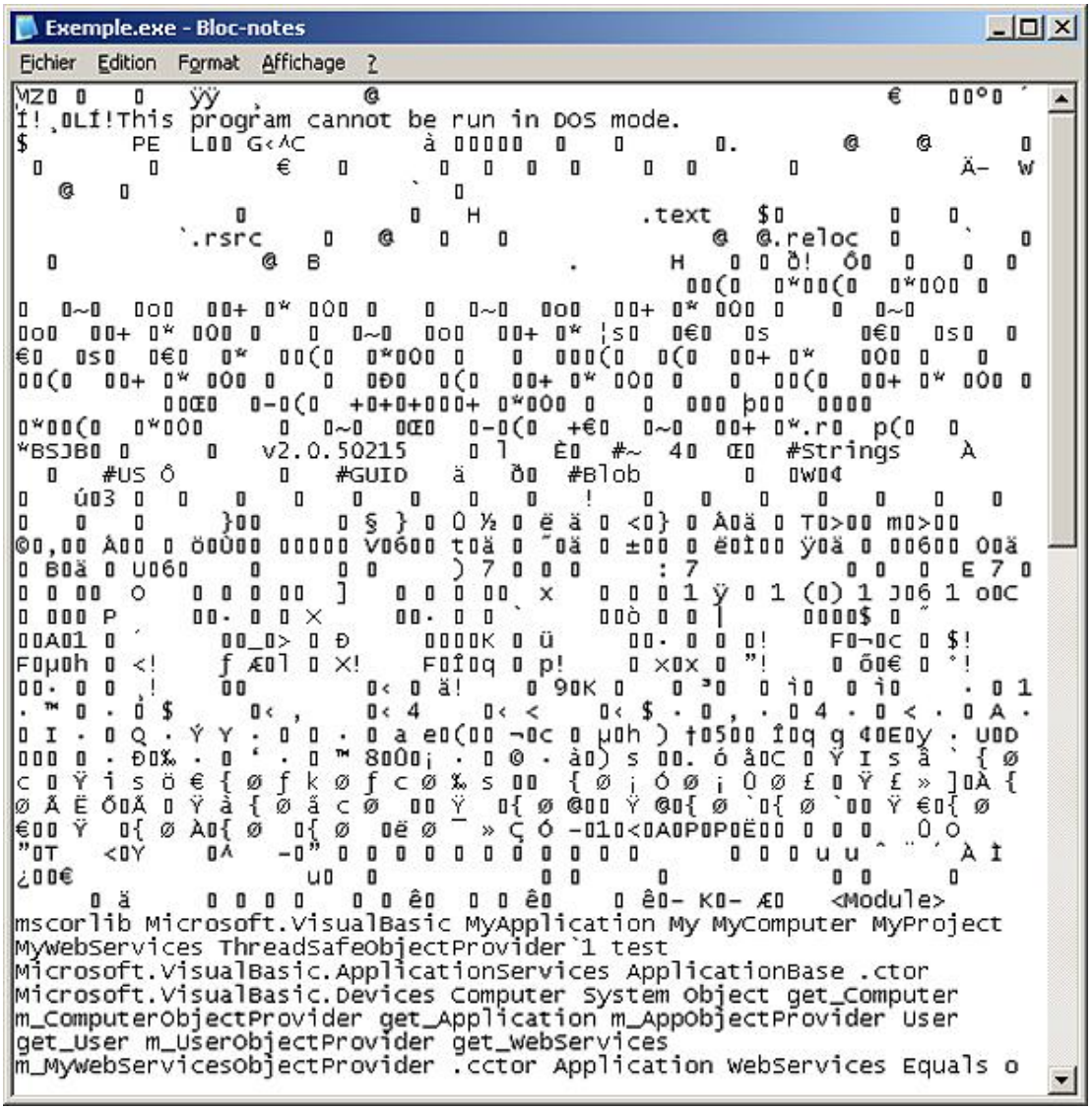
`/reference :liste de fichiers`

Cette option indique au compilateur la liste des fichiers référencés dans le code et nécessaires pour la compilation. Les noms des fichiers doivent être séparés par une virgule.

3. Analyse d'un fichier compilé

Maintenant que notre fichier exécutable est créé, essayons de voir ce qu'il contient.

Première solution : l'ouvrir avec le bloc-notes de Windows



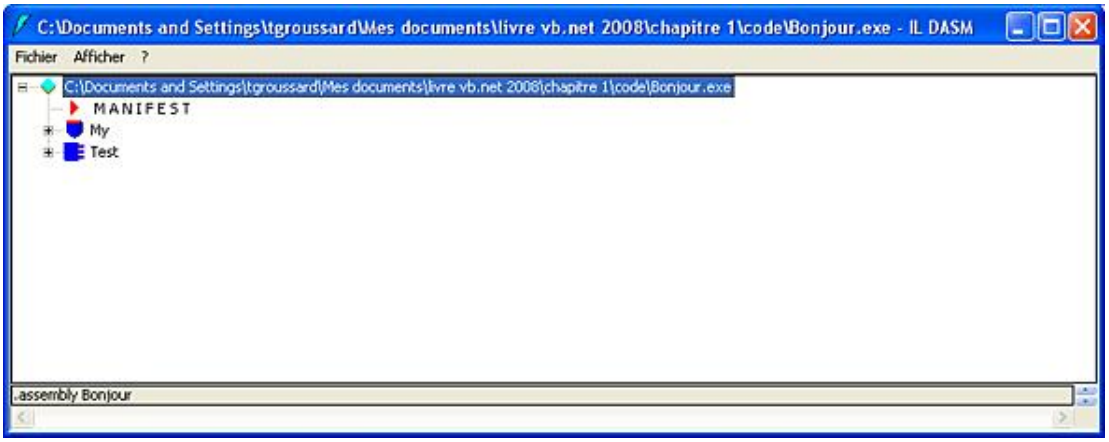
Le résultat n'est pas très parlant, c'est le moins que l'on puisse dire !

Nous avons dit que le compilateur génère du code MSIL. C'est donc ce code que nous visualisons dans le bloc-notes. Pour visualiser le contenu d'un fichier MSIL, le framework .NET propose un outil plus adapté.

Deuxième solution : utiliser un désassembleur

Cet outil est lancé à partir de la ligne de commande par l'instruction`ildasm`.

Il permet de visualiser, de manière plus claire que le bloc-notes, un fichier généré par le compilateur. Il convient d'indiquer le fichier que l'on souhaite examiner, par le menu **Fichier - Ouvrir**. Le désassembleur affiche alors son contenu.



Les informations présentes dans le fichier peuvent être séparées en deux catégories : le manifest et le code MSIL. Le

manifest contient les métadonnées permettant de décrire le contenu du fichier et les ressources dont il a besoin. On parle dans ce cas de fichier auto descriptif. Cette technique est très intéressante car dès que le Common Language Runtime lit le fichier, il dispose de toutes les informations nécessaires pour son exécution.

Il n'y a plus besoin d'avoir recours à un enregistrement dans le registre de la machine. Le manifest peut être visualisé par un double clic sur son nom.

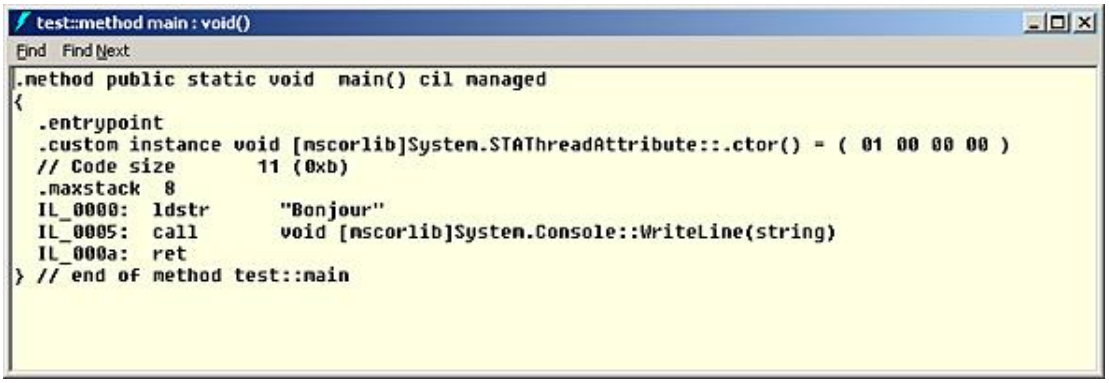
Nous retrouvons dans ce manifest des informations indiquant que, pour que l'application puisse fonctionner, elle a besoin des assemblages externes mscorlib, Microsoft.VisualBasic et System.

La deuxième partie correspond réellement au code MSIL. Un ensemble d'icônes est utilisé pour faciliter la visualisation des informations.

Symbole	**Signification**
	Plus d'infos
	Espace de noms
	Classe
	Interface
	Classe de valeurs
	Enumération
	Méthode
	Méthode statique
	Champ
	Champ statique
	événement

	Propriété
	élément de manifeste ou d'infos de classe

Comme pour le manifest, un double clic sur un élément permet d'obtenir plus de détails. Ainsi, nous pouvons, par exemple, visualiser la traduction de notre procédure main.

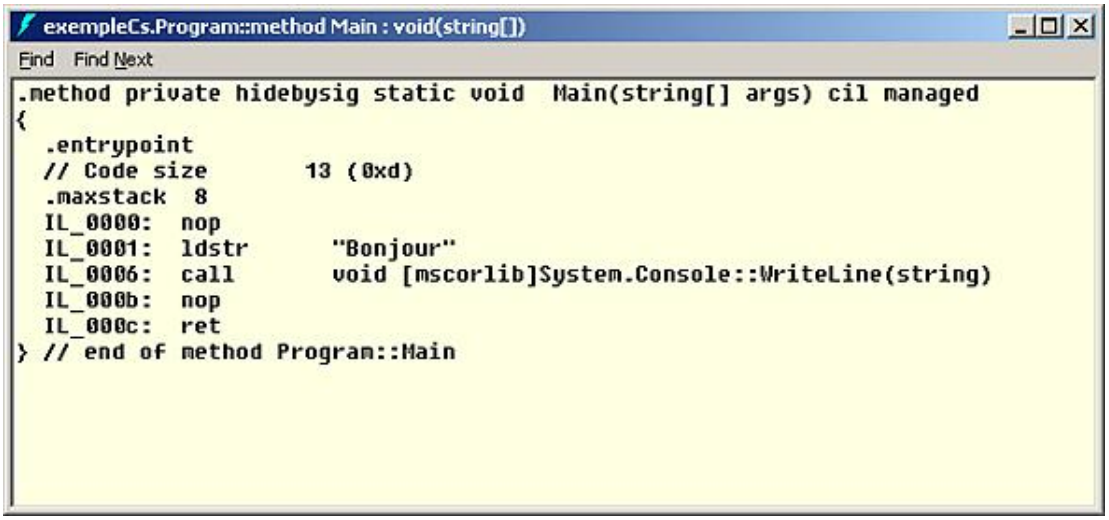


Dans un exemple de code aussi simple, il est facile de faire la correspondance entre le code Visual Basic et sa traduction en code MSIL. Pour les personnes enthousiasmées par le code MSIL, il existe un assembleur MSIL : `ilasm`. Cet outil attend un fichier texte contenant du code MSIL et le transforme en format binaire.

Puisque nous sommes capables de visualiser le code MSIL, nous pouvons vérifier qu'il est bien indépendant du langage source utilisé pour développer l'application. Voici donc le code C# qui réalise la même chose que notre code Visual Basic.

```
using System;
    class Program
    {
        static String message = "Bonjour";
        static void Main(string[] args)
        {
            Console.WriteLine(message);
        }
 }
```

Après compilation et désassemblage par `ildasm`, voici ce qu'il nous présente pour la méthode Main.



Il n'y a aucune différence par rapport à la version Visual Basic de la méthode Main.

Il est également possible de faire la démarche inverse en transformant un fichier texte contenant du code MSIL en fichier binaire correspondant. Cette transformation se fait grâce à l'assembleur ilasm. La seule difficulté est de créer un fichier texte contenant le code MSIL car même si la syntaxe est compréhensible, elle n'est pas très intuitive. Une solution peut être de demander à l'outil ildasm (le désassembleur) de générer ce fichier texte. Pour cela après, avoir ouvert le fichier exécutable ou la bibliothèque dll avec ildasm, vous devez utiliser l'option **Dump** du menu **Fichier**. Vous êtes alors invité à choisir le nom du fichier à générer (extension .il).

Ce fichier peut être ensuite modifié avec un simple éditeur de texte. Remplacez par exemple le contenu de la variable message avec la chaîne "Hello".

```
.method private hidebysig specialname rtspecialname static
         void  .cctor() cil managed
  {
    // Code size       11 (0xb)
    .maxstack  8
    IL_0000:  ldstr      "Hello"
    IL_0005:  stsfld     string Program::message
    IL_000a:  ret
  } // end of method Program::.cctor
```

Sauvegardez ensuite le fichier. Il ne reste plus maintenant qu'à regénérer le fichier exécutable grâce à l'assembleur ilasm. Saisissez pour cela la ligne de commande suivante :

```
ilasm Bonjour.il /output=Hello.exe
```

L'option `/output=Hello` permet d'indiquer le nom du fichier généré. Si cette option n'est pas spécifiée c'est le nom du fichier source qui sera utilisé. Vous pouvez maintenant lancer le nouvel exécutable et vérifier le message affiché. Toutes ces manipulations peuvent se faire sur n'importe quel fichier exécutable ou bibliothèque dll. La seule difficulté réside dans le volume d'informations fourni par la décompilation. Ceci pose cependant un problème : toute personne disposant des fichiers exécutables ou bibliothèques dll d'une application peut modifier l'application. Certes les modifications risquent d'être périlleuses mais la modification d'une valeur représentant une information importante pour l'application (mot de passe, clé de licence...) est envisageable. Une parade possible à ce genre de manipulation consiste à rendre le code généré par le décompilateur le plus incompréhensible possible. Pour cela, il faut agir au niveau du fichier exécutable ou de la bibliothèque dll en modifiant les informations qu'ils contiennent sans, bien sûr, en perturber le fonctionnement. Des outils appelés obfuscateurs sont capables de réaliser cette opération. Visual Studio est fourni avec un outil de la société PreEmptive Solutions appelé DotFuscator Community Edition. Cette version permet de réaliser les opérations de base pour "embrouiller" un fichier. Le principal traitement effectué sur le fichier consiste à renommer les identifiants contenus dans le fichier (nom des variables, nom des procédures et fonctions…) avec des valeurs très peu explicites, en général un caractère unique. Voici un extrait de la décompilation du fichier Bonjour.exe après traitement par Dofuscator Community Edition.

```
.class private auto ansi beforefieldinit a
       extends [mscorlib]System.Object
{
  .field private static string a
  .method private hidebysig static void  a (string[] A_0) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldsfld     string a::a
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method a::a

  .method public hidebysig specialname rtspecialname
         instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method a:: .ctor

  .method private hidebysig specialname rtspecialname static
         void  .cctor() cil managed
  {
    // Code size       11 (0xb)
    .maxstack  8
    IL_0000:  ldstr      "Bonjour"
    IL_0005:  stsfld     string a::a
    IL_000a:  ret
```

```
  } // end of method a:: .cctor

} // end of class a
```

Dans ce fichier plus aucune trace des noms utilisés dans le code. La procédure main s'appelle maintenant 'a', la variable message s'appelle maintenant 'a' aussi. Imaginez le résultat d'un tel traitement sur un fichier contenant plusieurs dizaines de variables et procédures !

La version Professional Edition permet également le cryptage des chaînes des caractères, la modification et l'ajout de code inutile pour complexifier les structures de contrôles (boucles, conditions…).

Ci-dessous un exemple de transformation extrait de la documentation de Dotfuscator.

Le code original :

```
public int CompareTo(Object o)
{
int n = occurrences - ((WordOccurrence)o).occurrences;
if (n == 0)
    {
    n = String.Compare(word, ((WordOccurrence)o).word);
    }
return(n);
}
```

Le code généré :

```
public virtual int _a(Object A_0) {
int local0;
int local1;
local0 = this.a - (c) A_0.a;
if (local0 != 0) goto i0;
goto i1;
while (true) {
return local1;
i0: local1 = local0;
}
i1: local0 = System.String.Compare(this.b, (c) A_0.b);
goto i0;
}
```

L'analyse de milliers de lignes de code de ce type risque de provoquer quelques migraines ! Il est donc préférable de conserver le code source original pour les modifications ultérieures. Plus d'informations sont disponibles sur le site http://www.preemptive.com/.

4. Exécution du code

Lorsqu'un utilisateur exécute une application managée, le chargeur de code du système d'exploitation charge le Common Language Runtime qui ensuite démarre l'exécution du code managé. Comme le processeur de la machine sur laquelle s'exécute l'application ne peut pas prendre en charge directement le code MSIL, Le Common Language Runtime doit le convertir en code natif.

Cette conversion ne concerne pas la totalité du code au chargement de l'application. Il convertit le code au fur et à mesure des besoins. La démarche adoptée est la suivante :

- Au chargement d'une classe, le Common Language Runtime remplace chaque méthode de la classe par un morceau de code demandant au compilateur JIT de le compiler en langage natif.
- Par la suite, lorsque la méthode est utilisée dans le code, la portion de code générée au chargement entre en action et compile en code natif la méthode.
- Le morceau de code demandant la compilation de la méthode est ensuite remplacé par le code natif généré.
- Les futurs appels de cette méthode se feront directement sur le code natif généré.

Évolution de Visual Basic 1 à Visual Basic .NET 2008

Depuis la version 1.0, sortie en 1991, jusqu'à la version 6.0, sortie en 1998, Visual Basic a subi de nombreuses évolutions. Visual Basic fut conçu à l'origine comme un langage simple permettant de développer rapidement une application sous Windows (comme le permettait GWBASIC sous MS-DOS). Cette simplicité d'utilisation repose en grande partie sur l'environnement de développement, qui masque les tâches fastidieuses de la création d'une application sous Windows.

De la version 1.0 à la version 3.0, on n'assiste à aucune révolution dans VB mais à des évolutions classiques pour un langage de programmation.

Avec l'apparition de la version 4.0 en 1996, VB entre dans la cour des grands avec une multitude d'évolutions :

- possibilité de créer des applications 32 bits (la prise en charge des applications 16 bits était encore assurée) ;
- création de DLL à partir de VB ;
- utilisation de DLL (écrites en VB ou tout autre langage) ;
- apparition de fonctionnalités objet dans VB (utilisation de classes).

Malgré ou peut-être à cause de toutes ces évolutions, la version 4.0 de VB n'était pas très stable.

Très rapidement, en 1997, Microsoft sort la version 5.0 de Visual Basic qui n'amène pas de grandes évolutions si ce n'est la disparition des applications 16 bits.

Les évolutions de la version 6.0, qui sort un an plus tard, portent essentiellement sur la méthode d'accès aux bases de données, avec le remplacement de DAO (*Data Access Object*) des versions précédentes par ADO (*Active Data Object*), qui devient d'ailleurs la méthode commune aux langages Microsoft pour l'accès aux données.

Cette version devra toutefois attendre le service pack 4 pour permettre un fonctionnement correct de certains contrôles d'accès aux données (le *Data Environment*).

Bien qu'étant générée en code natif par la compilation, une application VB a toujours besoin du module runtime pour pouvoir s'exécuter sur une machine (vbrun.dll) car, contrairement à des langages comme C++, VB n'utilise pas l'interface de programmation WIN32 pour appeler les fonctions du système d'exploitation.

La version suivante qui sort en 2002 apporte des changements radicaux dans Visual Basic. Cette version s'intègre dans la suite Visual Studio .Net reposant sur une nouvelle infrastructure pour la création et l'exécution d'applications sous Windows : le Framework .NET. Les principes de fonctionnement de cette infrastructure sont décrits dans le chapitre Présentation dela plate-forme .NET.

Les versions 2003 et 2005 suivent l'évolution du Framework.NET (version 1.1 puis 2.0) apportant toujours plus de fonctionnalités et d'outils facilitant et accélérant le développeent d'applications.

La version 2008 apporte également son lot de nouveautés. Parmi les plus remarquables :

- Possibilité de générer une application pour une version spécifique du framework (2.0, 3.0, 3.5).
- Prise en charge ou amélioration de la prise en charge de nouvelles technologies pour le développement WEB (AJAX, Java Script, CSS…).
- Intégration de Linq dans les langages Visual Basic et Visual C# permettant d'uniformiser l'accès aux données indépendamment de la source (objets, bases de données, fichier XML).
- Ajout d'un utilitaire de mappage objet/relationnel (O/R Designer).
- Création d'applications WPF optimisées pour Windows Vista.
- Possibilité de créer des états avec Report Designer (en remplacement de Crystal Report).

Installation et premier démarrage

1. Configuration nécessaire

Pour permettre un fonctionnement correct, Visual Studio nécessite une configuration minimale. Microsoft conseille les valeurs suivantes :

Composant	Minimum recommandé	Performances optimales
Processeur	Pentium 1,6 GHz ou équivalent	Pentium 2,2 GHz ou équivalent
RAM	384 Mo	1024 Mo
Espace disque	1 GB sur le disque systèmeet de 2,8 à 3,8 GB sur un autre disque	
Video	1024 x 768	1280 x 1024
Lecteur CD-Rom ou DVD	Indispensable	Indispensable
Système d'exploitation	Microsoft Windows XPMicrosoft Windows Server 2003Microsoft Windows Vista	Toute version ultérieure (XP, 2003) avec le dernier service pack disponible (SP2 pour XP, SP2 pour Windows 2003)

2. Procédure d'installation

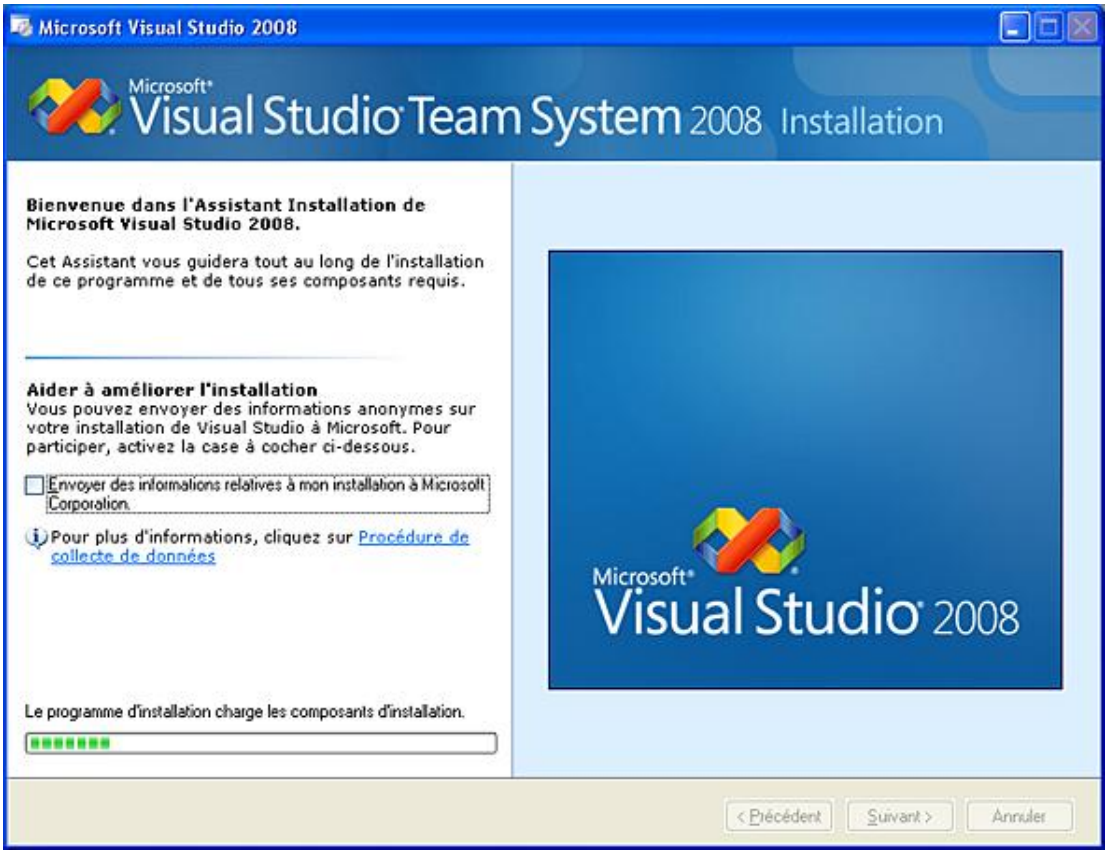
Les éléments nécessaires sont :

- les CD-Rom ou DVD de Visual Studio.NET ;
- de la place disponible sur votre disque dur (de 3,8 à 5 Go en fonction des outils installés) ;
- et surtout de la patience, car l'installation est longue...

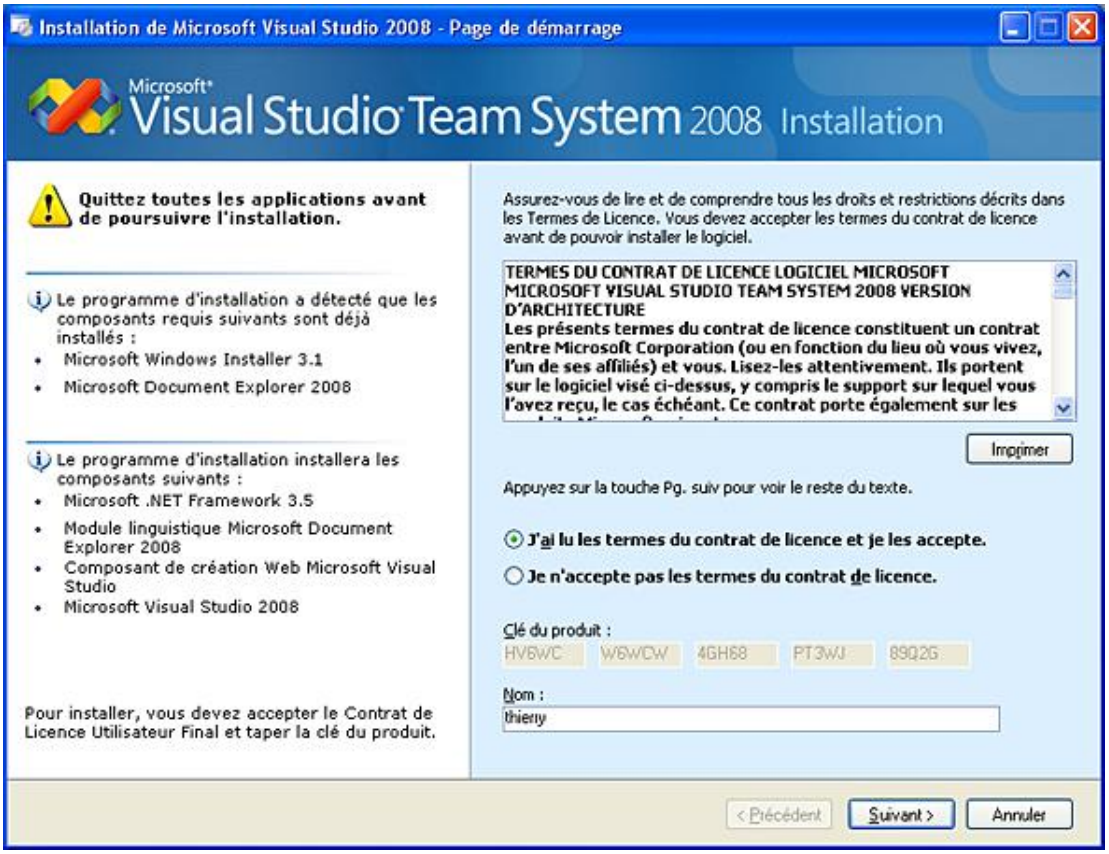
Après insertion du premier CD-Rom et quelques secondes de chargement, l'écran suivant s'affiche :

Cet écran propose, par des liens hypertexte, les trois actions nécessaires pour l'installation de Visual Studio. Nous devons bien sûr débuter par l'installation de Visual Studio.

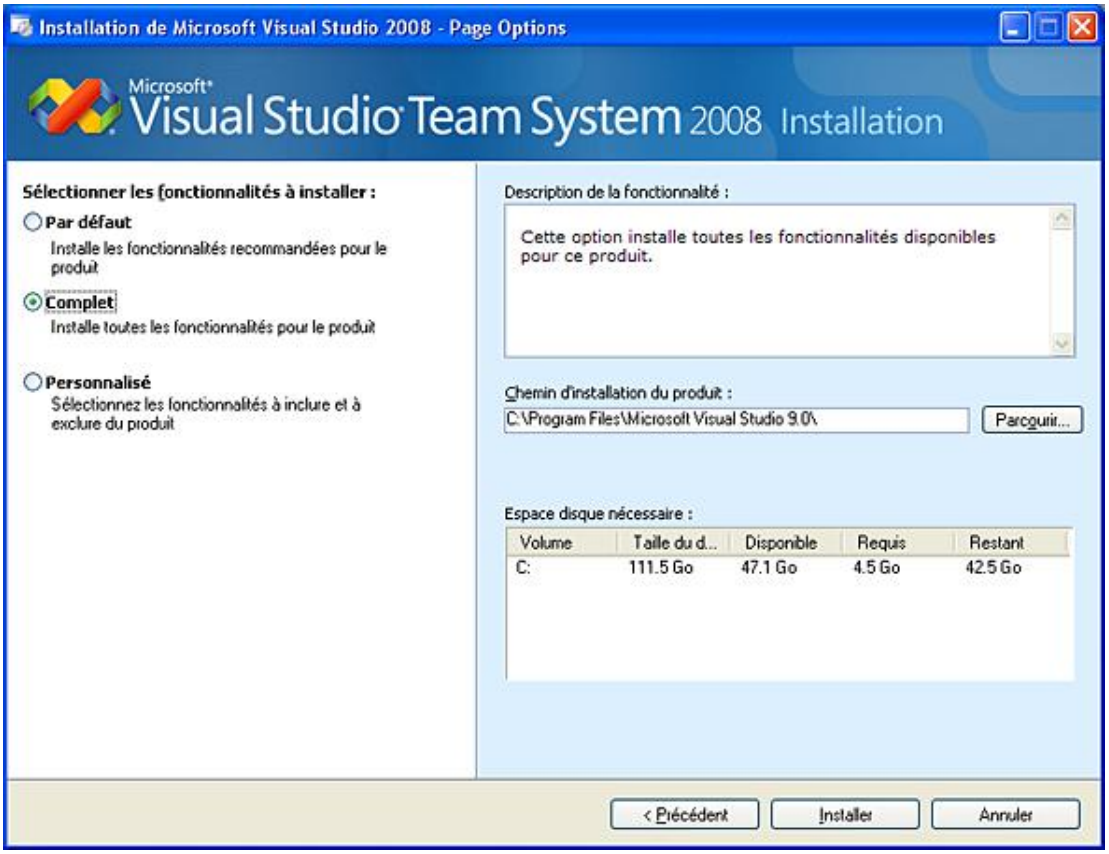
La première étape consiste, pour l'assistant d'installation, à collecter les informations concernant votre système :



L'écran suivant vous informe du résultat de l'analyse effectuée et vous demande d'accepter le contrat de licence et de saisir la clé associée à votre produit :

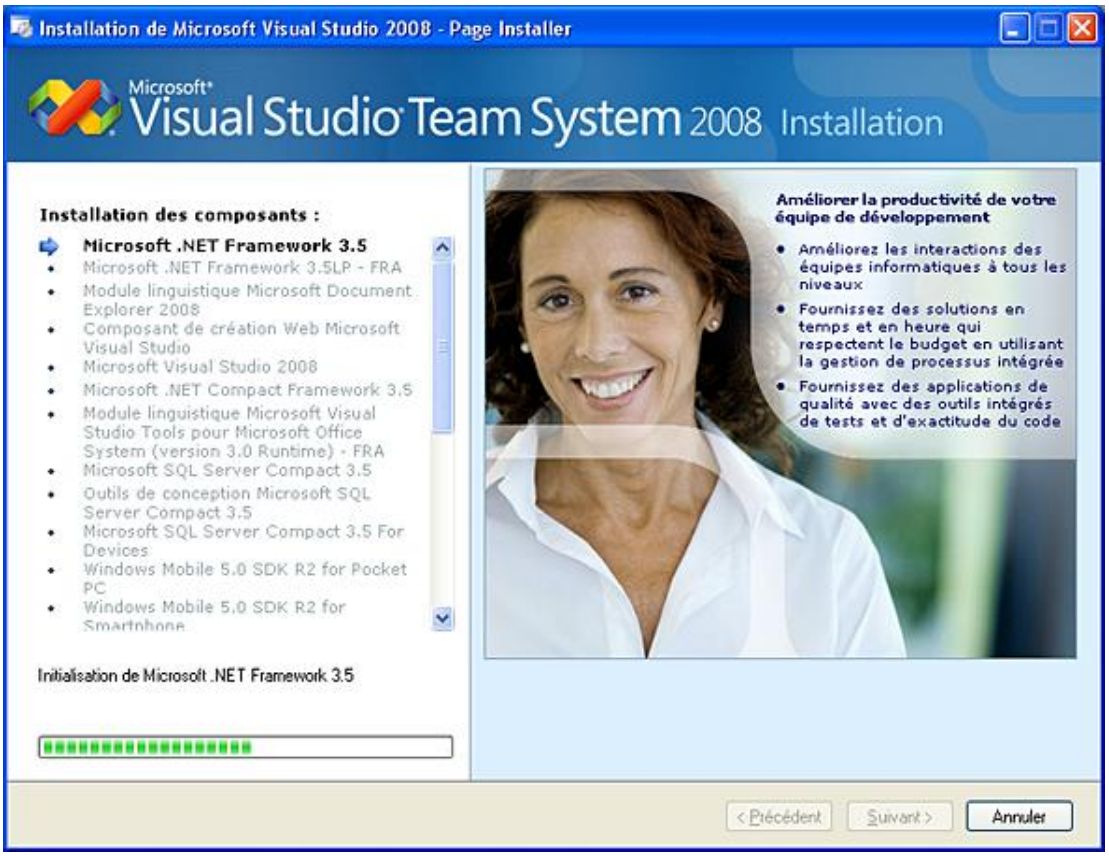


L'écran suivant vous propose de personnaliser l'installation en choisissant les outils et langages installés ainsi que le répertoire dans lequel ils seront installés. En fonction de vos choix, l'espace disque nécessaire pour l'installation est calculé. Le programme d'installation vérifie également que l'espace disque disponible est suffisant pour mener à bien l'installation.



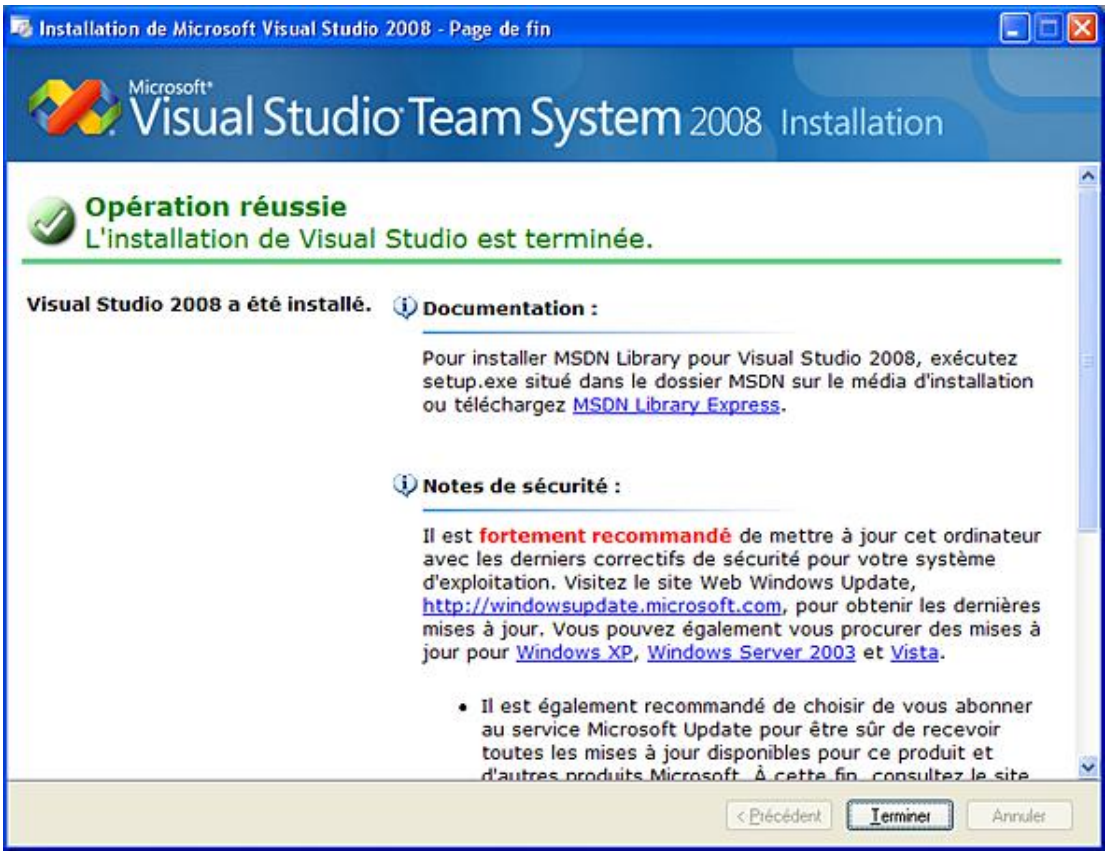
Après validation de vos paramètres par le bouton **Installer**, le programme d'installation débute la copie des fichiers.

L'écran suivant affiche la progression de l'installation :



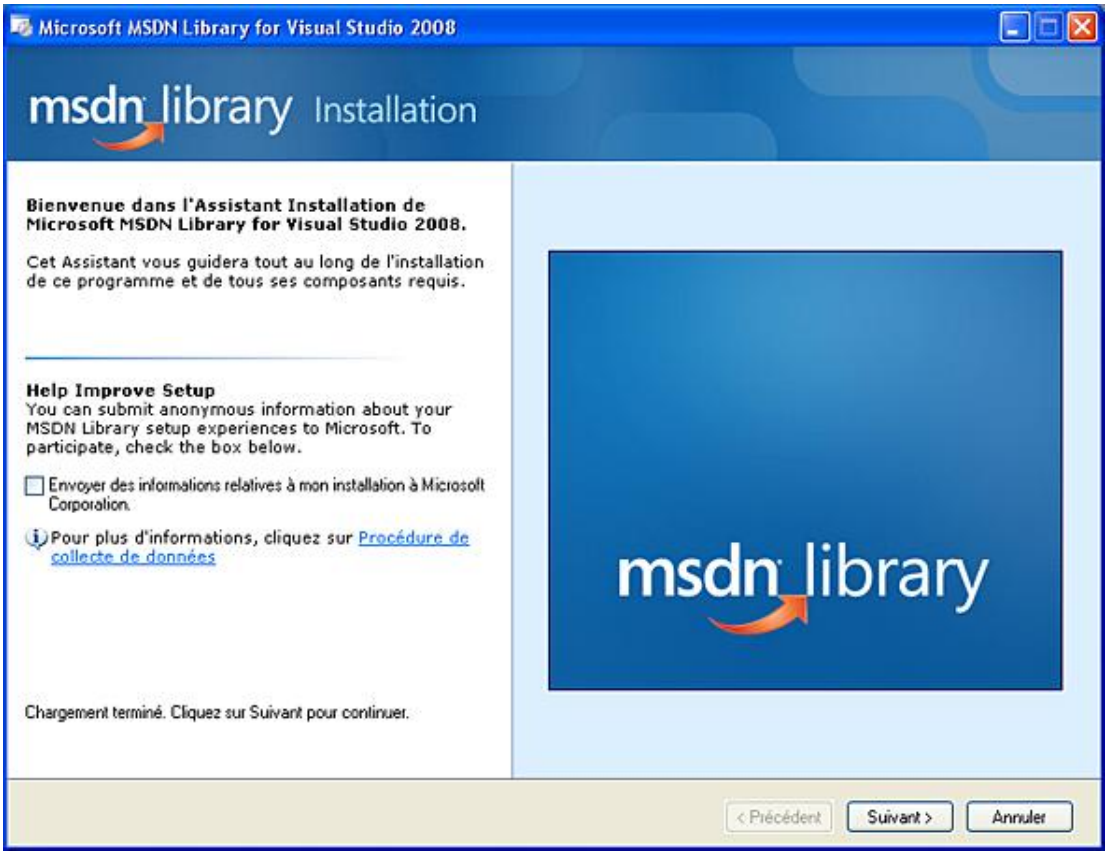
Si vous installez le produit à partir de CD-Rom, vous serez invité à insérer le disque suivant permettant de poursuivre la phase d'installation.

Après une trentaine de minutes de copie, l'écran suivant vous informe du succès de l'installation.

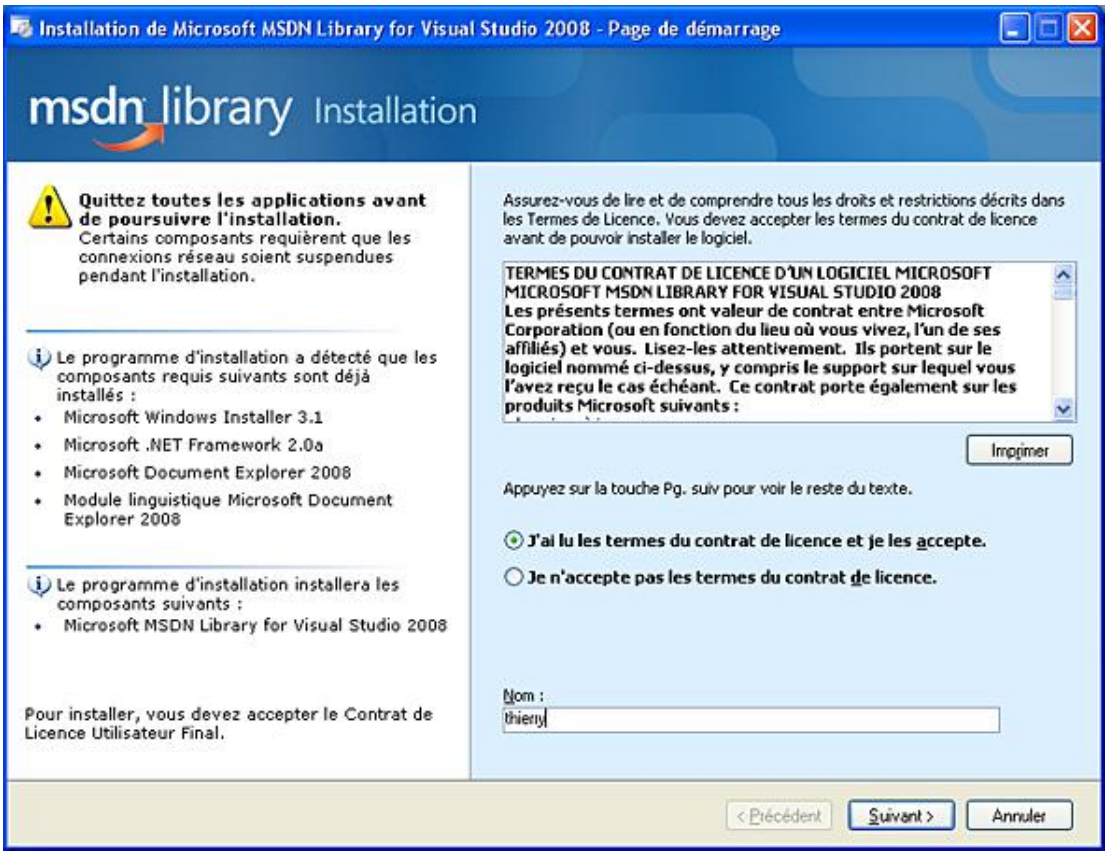


Après la fermeture de cet écran par le bouton **Terminer**, vous revenez à l'écran initial et pouvez alors choisir l'installation de la documentation MSDN.

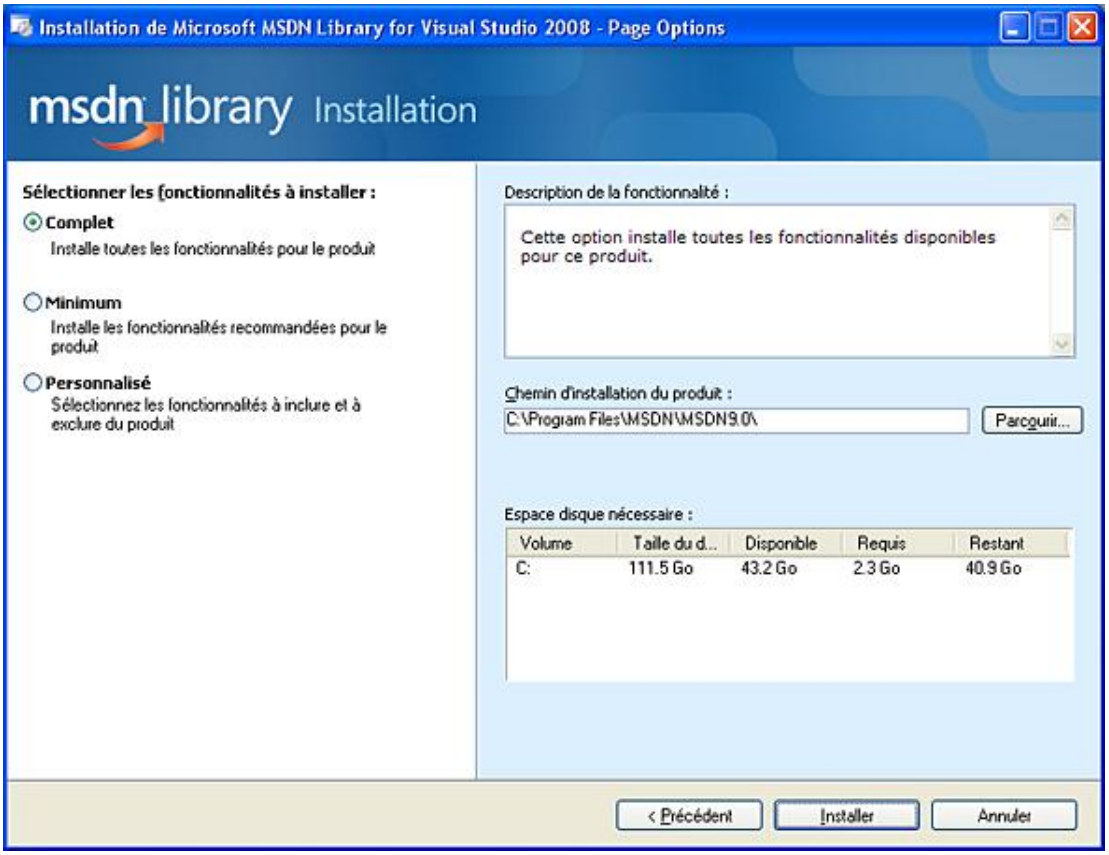
Comme pour l'installation de Visual Studio, un assistant vous guide dans les différentes étapes.



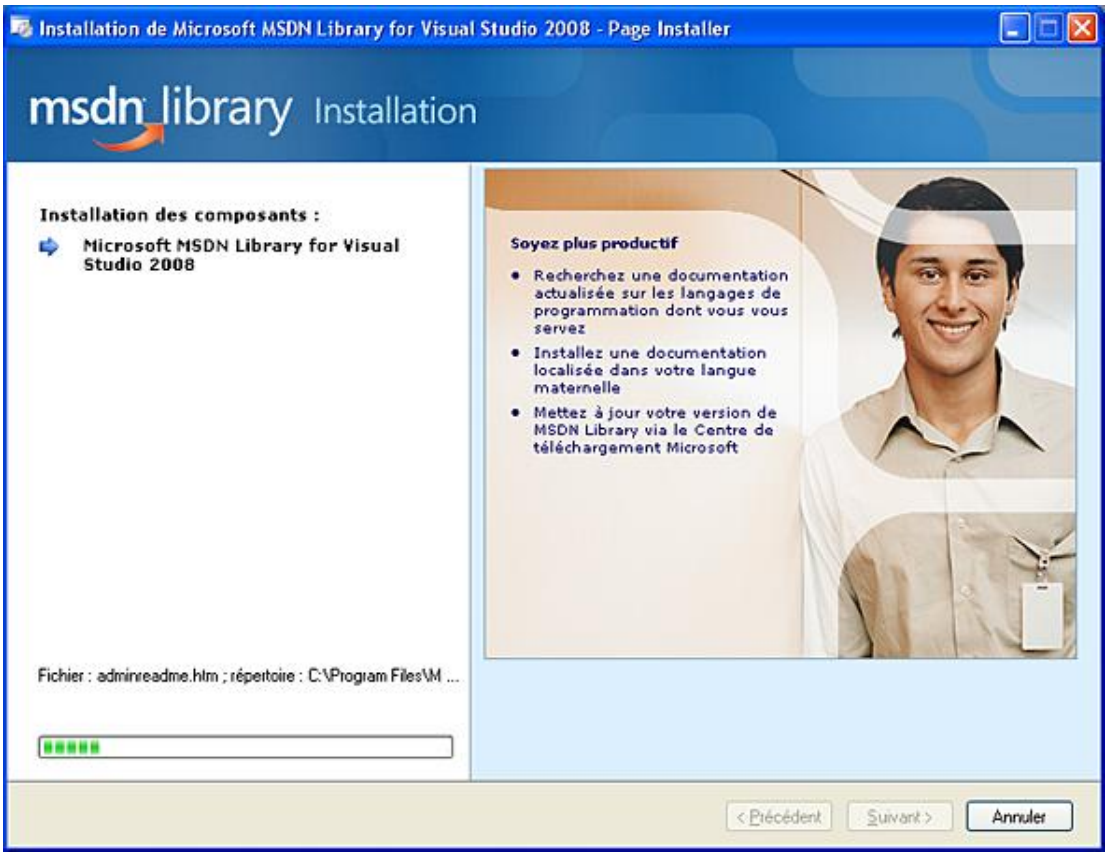
Vous devez accepter le contrat de licence MSDN pour pouvoir poursuivre l'installation.



Puis vous devez choisir le type d'installation désirée. Si vous ne choisissez pas l'installation complète, vous serez parfois obligé de fournir le ou les disques source de MSDN lors de la consultation de certaines rubriques d'aide. En fonction de vos choix, l'espace disque nécessaire est calculé et affiché.



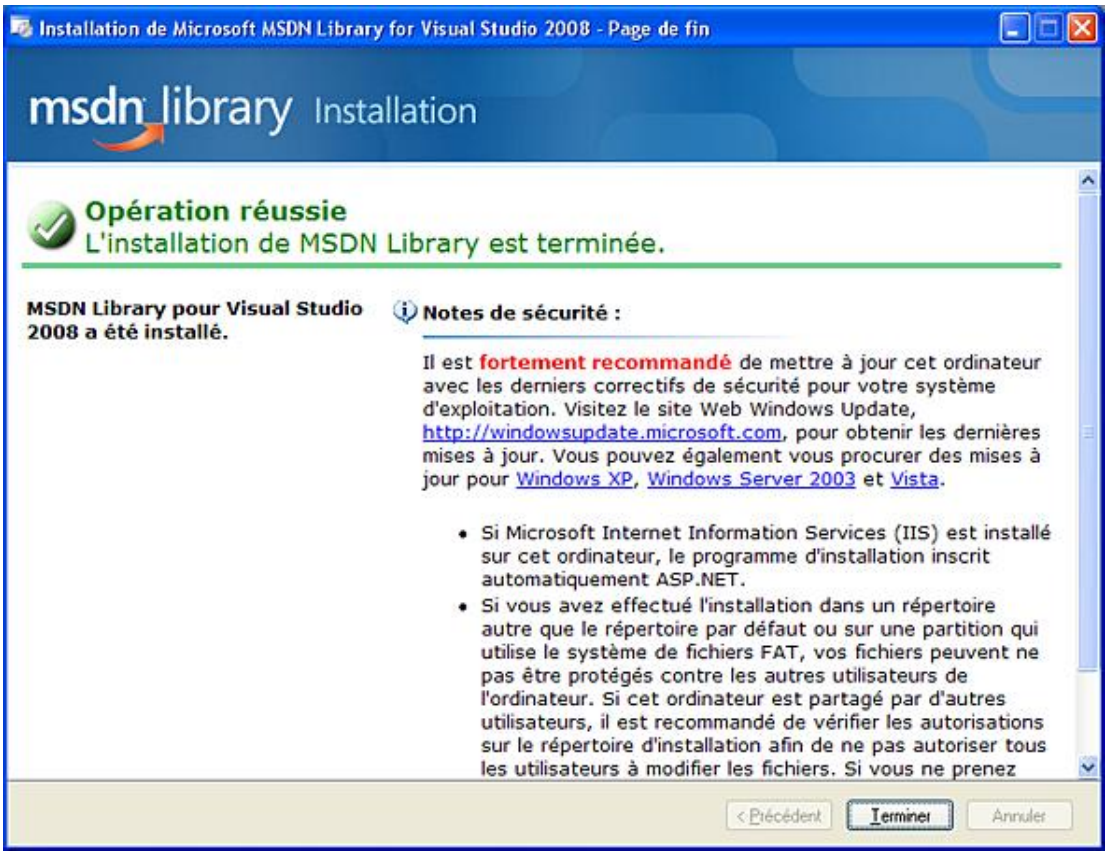
L'ultime étape démarre la copie des fichiers sur le disque.



En fonction de vos choix et du type de support utilisé pour l'installation, vous serez invité au cours de la copie à fournir

le ou les disques suivants.

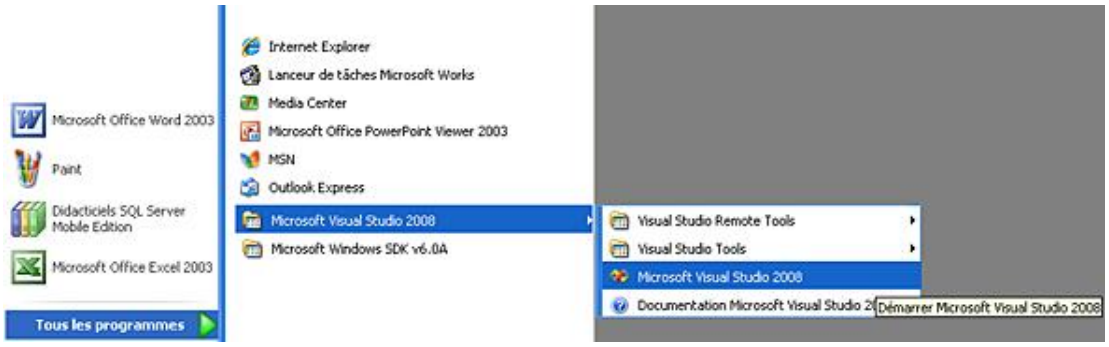
Après quelques minutes de copie, l'écran suivant vous confirme l'installation de la documentation MSDN.



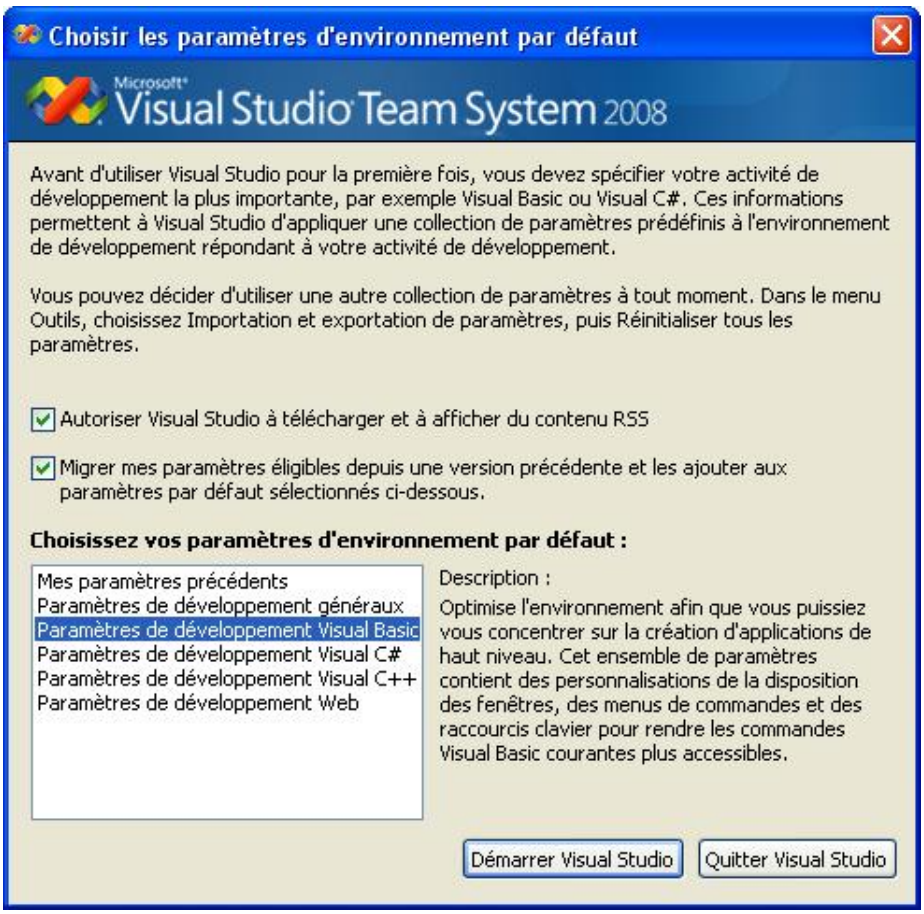
La dernière étape consiste à vérifier, auprès de Microsoft, s'il existe des correctifs pour les produits que vous venez d'installer. Cette étape nécessite un accès Internet avec bande passante suffisante car le volume d'informations à recevoir peut être important (Modem 56 K s'abstenir). Si vous ne disposez pas d'accès Internet, vous pouvez ignorer cette étape et tout de même disposer d'un produit opérationnel.

3. Premier démarrage

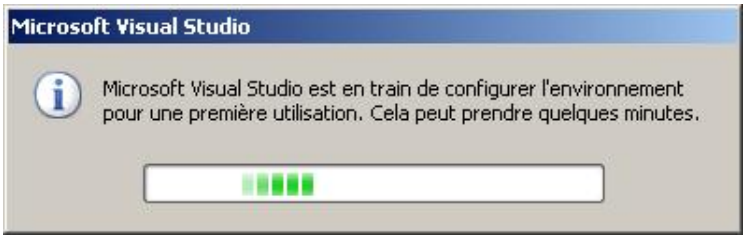
Un raccourci créé automatiquement par le programme d'installation vous permet de lancer Visual Studio.



Lors du premier démarrage, Visual Studio vous propose de personnaliser l'environnement de travail. En fonction de votre préférence pour un langage particulier, Visual Studio configure l'environnement avec les outils adaptés. Cette configuration peut, par la suite, être modifiée par le menu **Outils - Importation et exportation de paramètres**.



Visual Studio applique ensuite vos choix avant de démarrer.

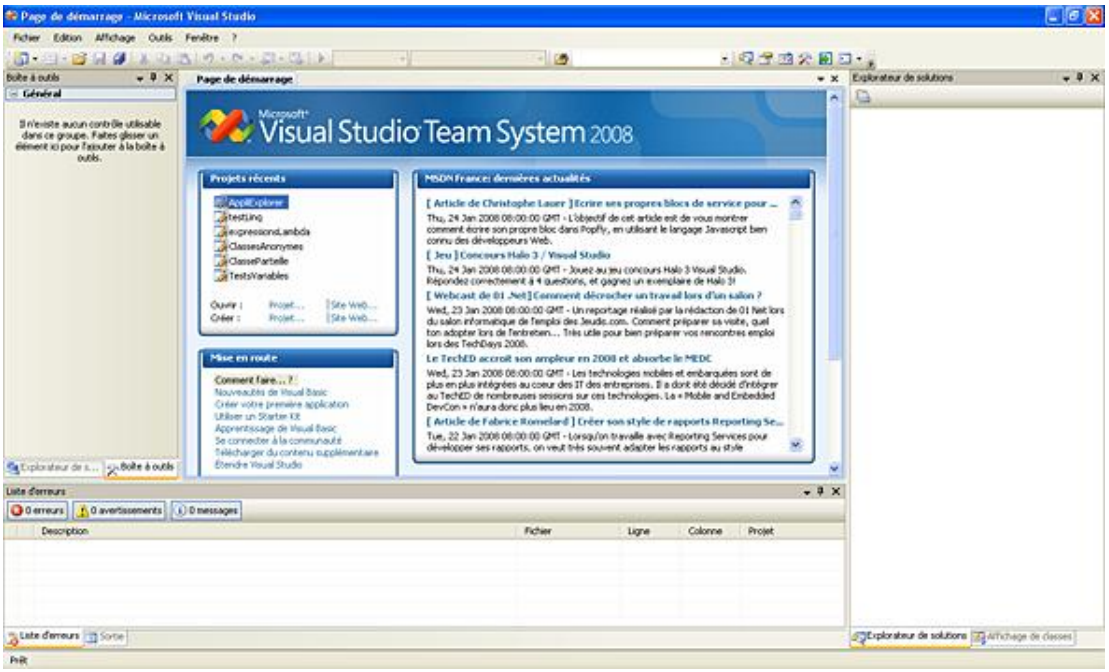


Nous allons maintenant explorer les outils à notre disposition.

Découverte de l'environnement

1. Page de démarrage

Cette page est affichée à chaque lancement de Visual Studio. Elle vous permet d'accéder rapidement aux derniers projets sur lesquels vous avez travaillé, de créer un nouveau projet ou d'ouvrir un projet existant. La rubrique **Mise en route** vous propose des liens vers les rubriques d'aide utiles pour une prise en main rapide de Visual Studio. Si une connexion Internet est disponible, la zone **Informations pour les développeurs Visual Studio** contient des informations en provenance du site Microsoft.



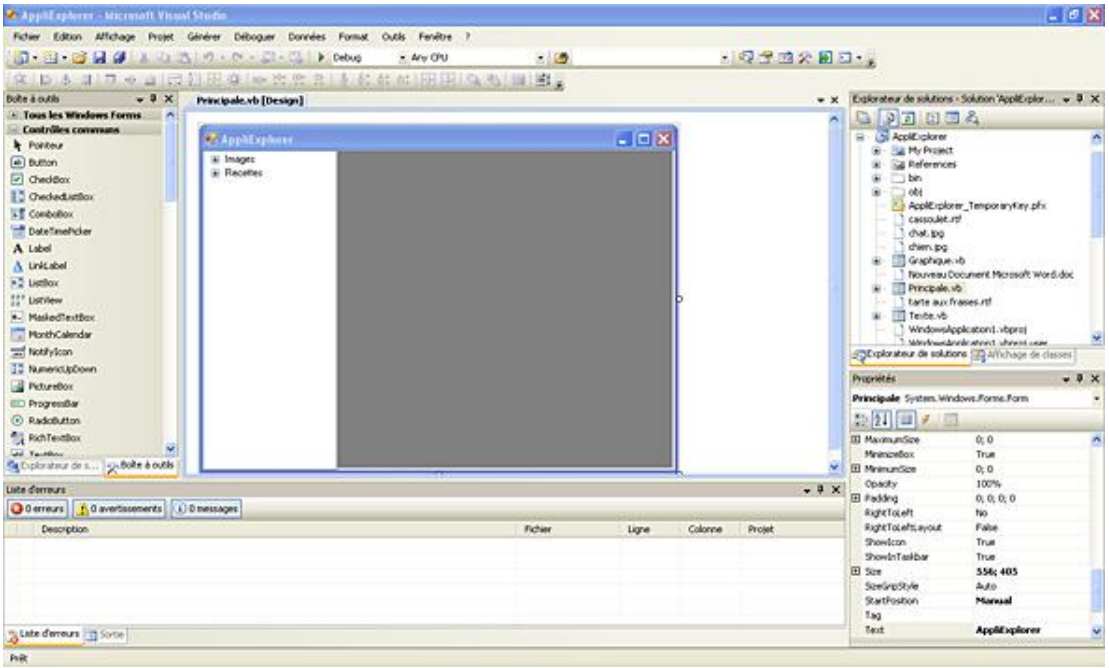
L'URL contactée pour renseigner cette rubrique est configurable par le menu **Outils - Options**. Après création d'un nouveau projet ou ouverture d'un projet existant, l'environnement Visual Studio est démarré.

2. Environnement Visual Studio

L'environnement est composé de trois types d'éléments :

- une zone de barre de menus et de barres d'outils ;
- une zone centrale de travail ;
- une multitude de fenêtres constituant les différents outils à notre disposition.

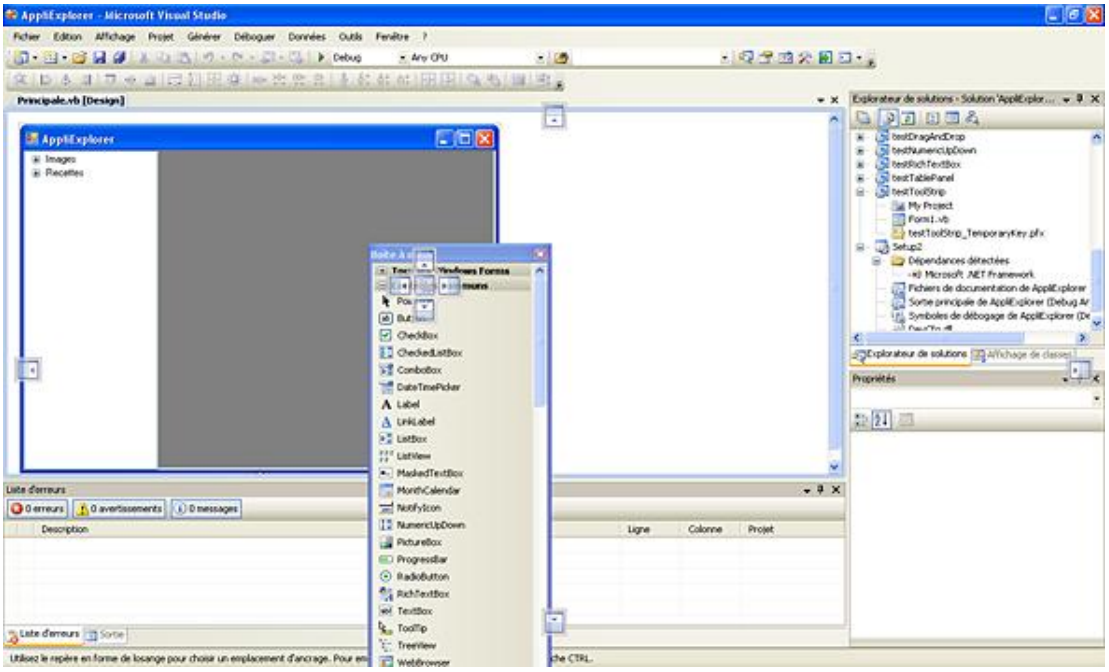
L'ensemble présente tout de même un aspect chargé et, après l'ajout d'une ou deux barres d'outils et l'apparition de quelques fenêtres supplémentaires, la zone de travail devient restreinte surtout sur un écran classique de 15 pouces.



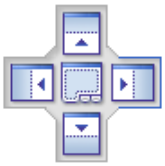
Heureusement, plusieurs solutions sont disponibles pour gérer notre espace de travail :

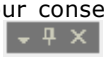
- l'ancrage des fenêtres ;
- le masquage automatique des fenêtres ;
- l'utilisation d'onglets.

L'ancrage de fenêtres ne permet pas de gagner de la place sur l'écran mais nous permet d'accrocher, à une bordure de l'écran ou d'une autre fenêtre, telle ou telle fenêtre. Il est également possible de rendre flottante chaque fenêtre, en double cliquant sur sa barre de titre ou en utilisant le menu contextuel. La fenêtre peut être ensuite déplacée ou ancrée sur une autre bordure. Pour nous guider dans l'ancrage d'une fenêtre, Visual Studio affiche, pendant le déplacement d'une fenêtre, des guides permettant de choisir la bordure où effectuer l'ancrage.



Les icônes  placées en périphérie de l'écran permettent l'ancrage sur la bordure correspondante de l'écran. Les

icônes  apparaissant au centre de la fenêtre survolée, contrôlent l'ancrage sur ses bordures ou sous forme d'un onglet supplémentaire pour la fenêtre.

Plus intéressantes pour gagner de la place sur l'écran, les fenêtres masquables ne sont visibles que si le curseur de la souris se trouve au-dessus de leur surface. Sinon, seule une zone d'onglets, située en bordure de l'environnement de développement, permet de faire apparaître son contenu. Pour conserver une fenêtre toujours visible, il suffit de la bloquer en utilisant la punaise présente sur sa barre de titre  .

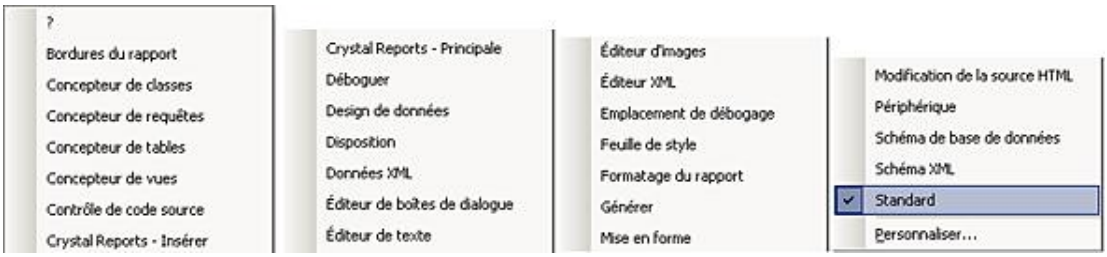
Enfin, l'utilisation d'onglets permet de partager une même zone écran entre différentes fenêtres et à ce niveau, les concepteurs de Visual Studio en ont usé sans modération.

Les outils disponibles

Regardons un peu plus en détail les différentes barres d'outils et fenêtres qui sont à notre disposition.

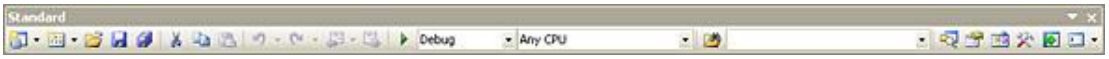
1. Les barres d'outils

Pas moins de trente barres d'outils différentes sont disponibles dans Visual Studio. L'affichage de chacune d'elles peut être contrôlé par le menu contextuel, obtenu en faisant un clic droit sur la barre de menus principale.



Il est bien sûr inutile d'afficher la totalité des barres d'outils simultanément mais seulement les plus utiles.

Standard



Éditeur de texte

Éditeur de boîtes de dialogue

Disposition

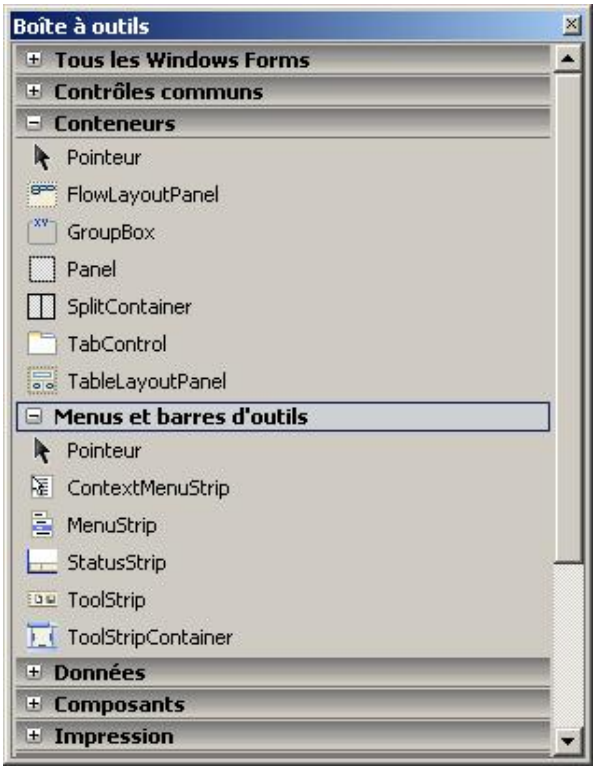
Déboguer

Les autres barres disponibles seront affichées, au coup par coup, en fonction de vos besoins afin d'éviter de surcharger votre écran.

Les fenêtres disponibles sont également assez nombreuses et nous allons découvrir les plus courantes.

2. La boîte à outils

C'est à partir de la boîte à outils que nous allons choisir les éléments utilisés pour la conception de l'interface de l'application.



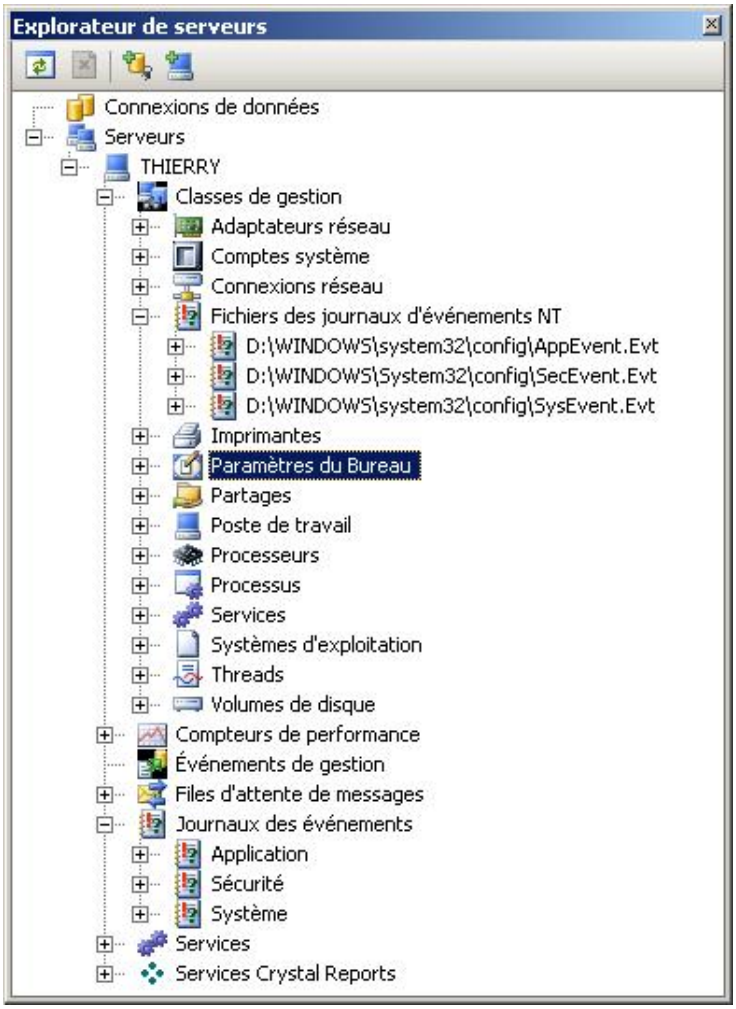
La boîte à outils est organisée par rubrique permettant de facilement retrouver les contrôles.

Chacun pourra personnaliser sa boîte à outils en y ajoutant par exemple, des contrôles non disponibles par défaut. Il peut être judicieux, avant d'ajouter des contrôles à votre boîte à outils, de créer une nouvelle rubrique pour les héberger. Pour cela, affichez le menu contextuel de la boîte à outils (en cliquant avec le bouton droit de la souris sur la boîte à outils), choisissez l'option **Ajouter un onglet** puis donnez un nom à la nouvelle rubrique ainsi créée. Après avoir sélectionné cette nouvelle rubrique, vous pouvez ensuite y ajouter des contrôles. Affichez à nouveau le menu contextuel de la boîte à outils puis choisissez l'option **Choisir les éléments**.

La liste des contrôles (Com ou .NET), disponibles sur la machine, est alors présentée, vous permettant ainsi de sélectionner les contrôles à ajouter dans cette rubrique de la boîte à outils. La configuration de la boîte à outils n'est pas liée au projet actif mais à l'environnement lui-même (la boîte à outils sera identique quel que soit le projet ouvert).

3. L'explorateur de serveurs

L'explorateur de serveurs est disponible par le menu **Affichage - Explorateur de serveurs** ou par le raccourci-clavier [Ctrl] [Alt] **S**. Il s'affiche sur un nouvel onglet de la fenêtre associée à la boîte à outils.



La majorité des applications a besoin pour fonctionner d'autres machines présentes sur le réseau. Il est donc nécessaire d'avoir, pendant la phase de développement d'une application, la possibilité d'accéder aux ressources disponibles sur d'autres machines.

L'élément le plus fréquemment utilisé de la fenêtre explorateur de serveurs sera certainement la rubrique **Connexions de données**.

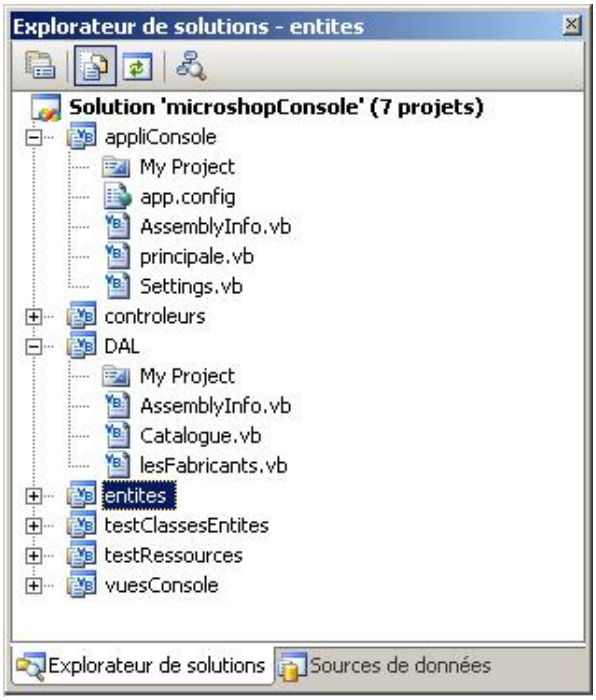
Elle permet notamment la gestion des objets disponibles sur le serveur SQL (tables, vues, procédures stockées).

L'explorateur de serveurs permet également de gérer les services fonctionnant sur les machines aussi bien par l'interface graphique que par le code. Il offre la possibilité de visualiser l'activité des machines en analysant les compteurs de performance ou en récupérant les informations dans les différents journaux d'événements. Un simple glisser-déplacer entre l'explorateur de serveurs et une fenêtre en cours de conception génère automatiquement le code permettant de manipuler cet élément dans l'application. Par exemple, le déplacement d'un compteur de performance au-dessus d'une fenêtre génère le code suivant :

```
Friend WithEvents perfCptMemoire As System.Diagnostics.PerformanceCounter
Me.perfCptMemoire = New System.Diagnostics.PerformanceCounter
Me.perfCptMemoire.CategoryName = "Mémoire"
Me.perfCptMemoire.CounterName = "Kilo-octets disponibles"
Me.perfCptMemoire.MachineName = "portableTG"
```

4. L'explorateur de solutions

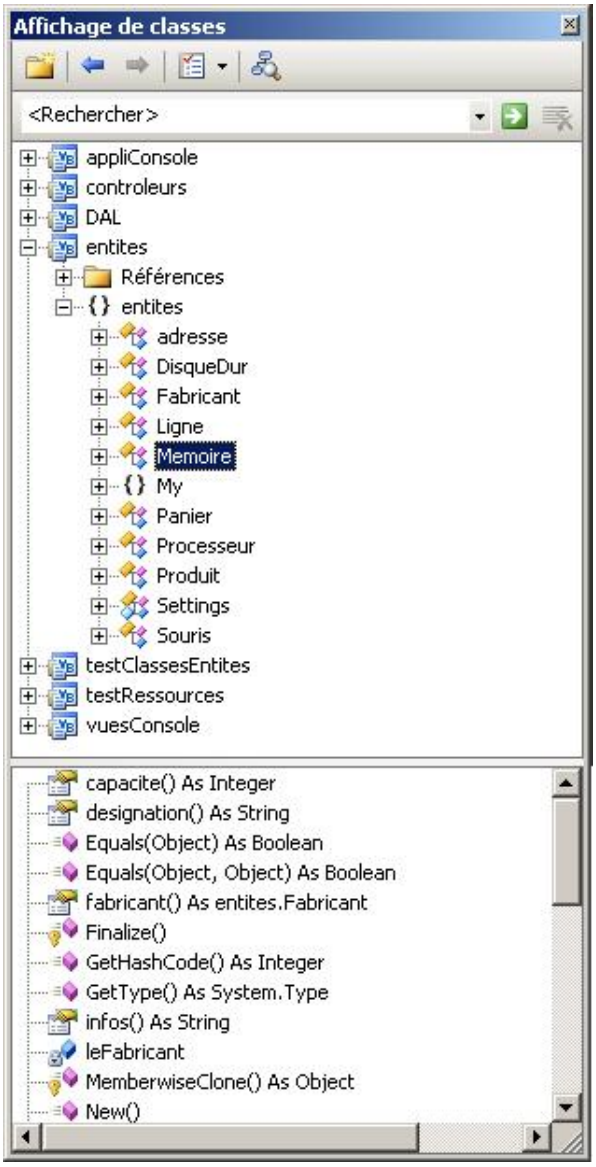
L'explorateur de solutions permet l'affichage des éléments constituant une solution et la modification de leurs propriétés.



L'utilisation de l'explorateur de solutions est présenté en détail dans le chapitre consacré à l'organisation d'une application.

5. L'affichage de classes

L'affichage de classes est accessible par le menu **Affichage - Autres fenêtres - Affichage de classes** ou par le raccourci-clavier [Ctrl] [Shift] **C**. Il partage sa zone écran avec l'explorateur de solutions.

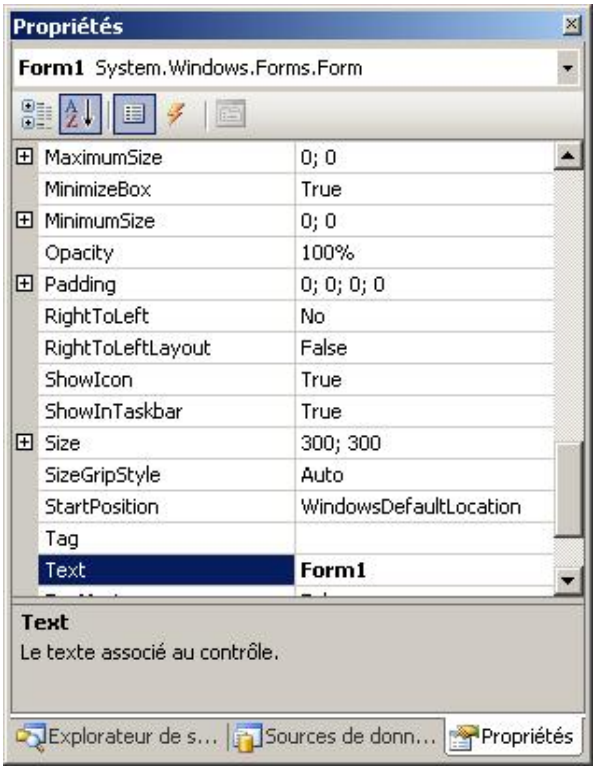


L'affichage de classe permet d'avoir une vision logique d'une solution en présentant les différentes classes utilisées dans cette solution.

6. La fenêtre de propriétés

La fenêtre de propriétés peut être affichée par trois méthodes :

- En utilisant le menu **Affichage - Fenêtre de propriétés**.
- Par la touche de fonction [F4].
- Par l'option **Propriétés** du menu contextuel disponible en cliquant avec le bouton droit sur un des éléments constituant un projet (élément graphique de l'interface utilisateur, fichier ou dossier du projet…). La fenêtre de propriétés adapte automatiquement son contenu en fonction de l'élément sélectionné et permet de modifier ces caractéristiques.



Les éléments dont vous pouvez modifier les caractéristiques peuvent être sélectionnés directement dans la liste déroulante ou sur l'interface de l'application.

Deux présentations sont disponibles pour la liste des propriétés :

Le mode **Alphabétique** que vous activez en cliquant sur l'icône  .

Le mode **Par catégorie** que vous activez en cliquant sur l'icône  .

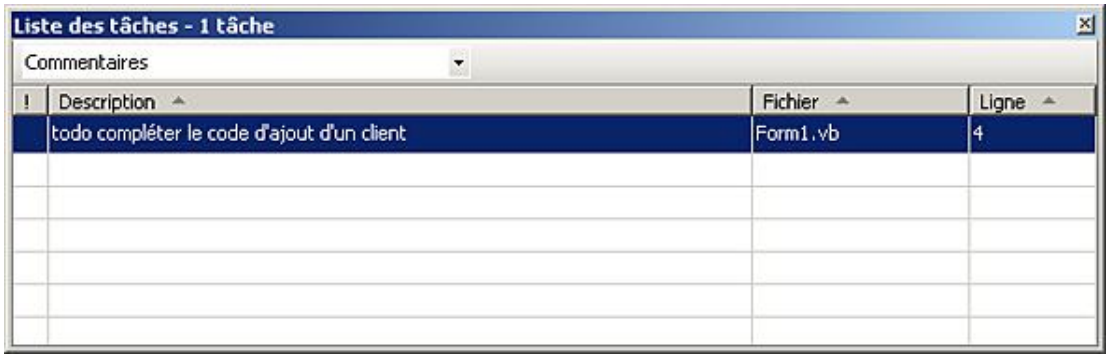
7. L'aide dynamique

L'aide dynamique de Visual Studio sélectionne les rubriques d'aide qui pourraient vous être utiles en fonction de votre activité ou des commandes et outils utilisés.

Dans l'exemple ci contre, un bouton de commande vient juste d'être sélectionné sur l'interface. Les différentes rubriques proposées sont donc centrées sur l'objet bouton de commande.

8. La liste des tâches

Cette fenêtre va vous permettre de remplacer les dizaines de PostIt collés sur le bord de votre écran. En effet, vous pourrez gérer ce qu'il reste à faire dans votre projet en tenant à jour une liste des modifications à apporter dans votre code.



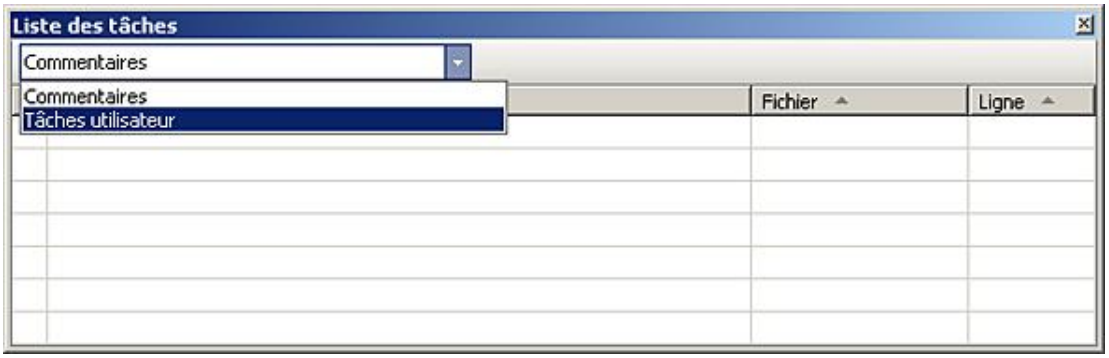
Les informations présentes dans la liste peuvent avoir deux origines :

- Les commentaires insérés dans votre code.
- Les informations saisies directement dans la fenêtre.

Vous pouvez placer dans votre code des commentaires qui apparaîtront par la suite dans la liste des tâches. Cette technique vous permet par exemple d'indiquer une modification à effectuer plus tard dans votre code.

Il suffit que le commentaire commence par todo, pour être ensuite repris automatiquement dans la liste des tâches.

Vous pouvez également saisir directement les informations dans la liste des tâches. Vous devez basculer vers l'affichage des tâches utilisateur en utilisant la zone de liste disponible sur la barre de titre de la liste des tâches.



L'ajout d'une tâche s'exécute ensuite par le bouton  , disponible dans la liste des tâches.

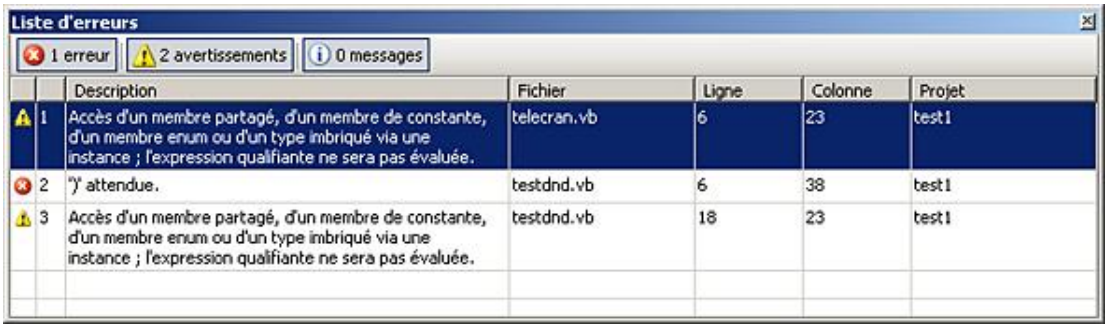
Vous pouvez alors spécifier une description et une priorité pour la nouvelle tâche en cliquant sur la colonne de gauche de la liste des tâches. Trois niveaux de priorité sont disponibles :

- Haute
- Normale
- Basse.

Pour chaque tâche, une case à cocher permet d'indiquer qu'elle a été réalisée. Sa description apparaît alors barrée dans la liste des tâches. Il n'y a pas, pour les tâches utilisateur, de liaison automatique avec une portion quelconque de code.

9. La liste des erreurs

Le code que vous saisissez est analysé en continu par Visual Studio et les éventuelles erreurs de syntaxe sont reprises par Visual Studio dans la fenêtre **Liste d'erreurs**.



Pour aller directement sur la ligne où une erreur de syntaxe est apparue, il suffit de double cliquer dans la liste sur l'élément correspondant (dans l'exemple précédent, double cliquer sur " ')' attendue " pour atteindre la ligne 6). Vous n'avez donc nul besoin de demander la compilation complète du code pour traquer toutes les erreurs de syntaxe. Dès que l'erreur est corrigée, elle disparaît automatiquement de la liste des erreurs.

Les boutons erreur, avertissement, message activent un filtrage sur les messages affichés dans la liste des erreurs.

10. La fenêtre d'édition de code

C'est certainement dans cette fenêtre que nous allons passer le plus de temps. Elle propose de nombreuses fonctionnalités permettant d'automatiser les actions les plus courantes.

a. Les Snippets

Les Snippets sont des morceaux de code qui peuvent très facilement être incorporés dans un fichier source. Ils permettent d'écrire très rapidement des portions de code correspondant à des situations courantes. Visual Studio propose, de base, une multitude de Snippets. Trois solutions sont disponibles pour insérer un Snippet :

- Utilisez l'option**Insérer un extrait** du menu contextuel de l'éditeur de code.
- Utilisez les combinaisons de touches [Ctrl] **K** puis [Ctrl] **X**.
- Saisissez le nom du Snippet puis la touche [Tab].

Pour les deux premières méthodes, Visual Studio vous propose de choisir dans une liste le Snippet qui vous intéresse. Certaines portions de code du Snippet peuvent être personnalisées. Ces portions de code sont surlignées en bleu clair. La modification d'une de ces portions de code répercute le changement sur toutes les occurrences dans le Snippet.

Dans l'exemple suivant, un Snippet a été utilisé pour ajouter une nouvelle propriété à une classe.

```
    Private newPropertyValue As Integer
    Public Property NewProperty() As Integer
        Get
            Return newPropertyValue
        End Get
        Set(ByVal value As Integer)
            newPropertyValue = value
        End Set
    End Property
```

La modification des valeurs newPropertyValue, Integer et NewProperty sera effectuée en cascade sur l'ensemble du code du Snippet.

Vous pouvez également concevoir vos propres Snippets. Vous devez pour cela créer un fichier XML qui va contenir le code du Snippet. Ce fichier doit avoir l'extension .snippet.

Pour vous aider dans la création d'un Snippet, Microsoft a prévu un Snippet. Vous pouvez l'incorporer dans votre fichier XML par le menu contextuel **Insérer un extrait - Snippet**.

Vous devez alors obtenir le document suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippet Format="1.0.0"
xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <Header>
    <Title>title</Title>
    <Author>author</Author>
    <Shortcut>shortcut</Shortcut>
    <Description>description</Description>
    <SnippetTypes>
      <SnippetType>SurroundsWith</SnippetType>
      <SnippetType>Expansion</SnippetType>
    </SnippetTypes>
  </Header>
  <Snippet>
    <Declarations>
      <Literal>
        <IDname/ID>
        <Defaultvalue/Default>
      </Literal>
    </Declarations>
    <Code Language="XML">
      <![CDATA[<test>
      <name>$name$</name>
      $selected$ $end$</test>]]>
    </Code>
  </Snippet>
</CodeSnippet>
```

Avant de voir comment compléter ce fichier, vous devez apporter une petite modification à sa structure. Les trois premières lignes doivent être modifiées afin d'obtenir la forme suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<CodeSnippets
xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
```

Vous devez également ajouter la balise de fermeture suivante à la fin du fichier.

```
</CodeSnippets>
```

Vous pouvez ensuite personnaliser votre Snippet. Dans un premier temps, vous devez modifier la section Header en remplaçant les valeurs des différentes balises.

```
  <Header>
    <Title>Parcours d'un tableau</Title>
    <Author>Thierry</Author>
    <Shortcut>tablo</Shortcut>
    <Description>ce Snippet ajoute une boucle permettant de parcourir un
 tableau</Description>
    <SnippetTypes>
      <SnippetType>Expansion</SnippetType>
    </SnippetTypes>
  </Header>
```

La section Déclarations permet ensuite de créer les paramètres utilisés dans le Snippet. Pour chaque paramètre, vous devez créer une section `<Literal>` et fournir un nom pour le paramètre et une valeur par défaut.

```
<Declarations>
  <Literal>
    <ID>nomtableau</ID>
    <Default>leTableau</Default>
  </Literal>
  <Literal>
    <ID>typeTableau</ID>
    <Default>TypeDuTableau</Default>
  </Literal>
```

```
  <Literal>
    <ID>tailleTableau</ID>
    <Default>tailleDuTableau</Default>
  </Literal>
</Declarations>
```

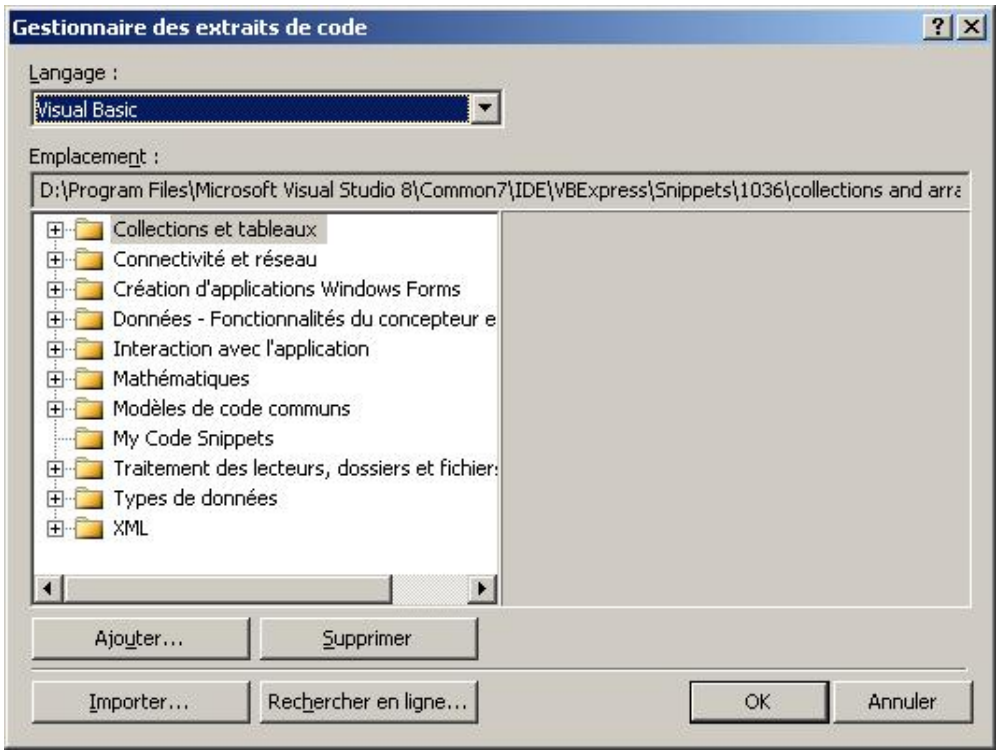
Vous devez ensuite indiquer pour quel langage votre Snippet est prévu.

```
<Code Language="VB">
```

Puis, enfin, définir dans la balise CDATA le code du Snippet. Dans ce code, vous pouvez utiliser les paramètres du Snippet en les encadrant entre deux caractères $.

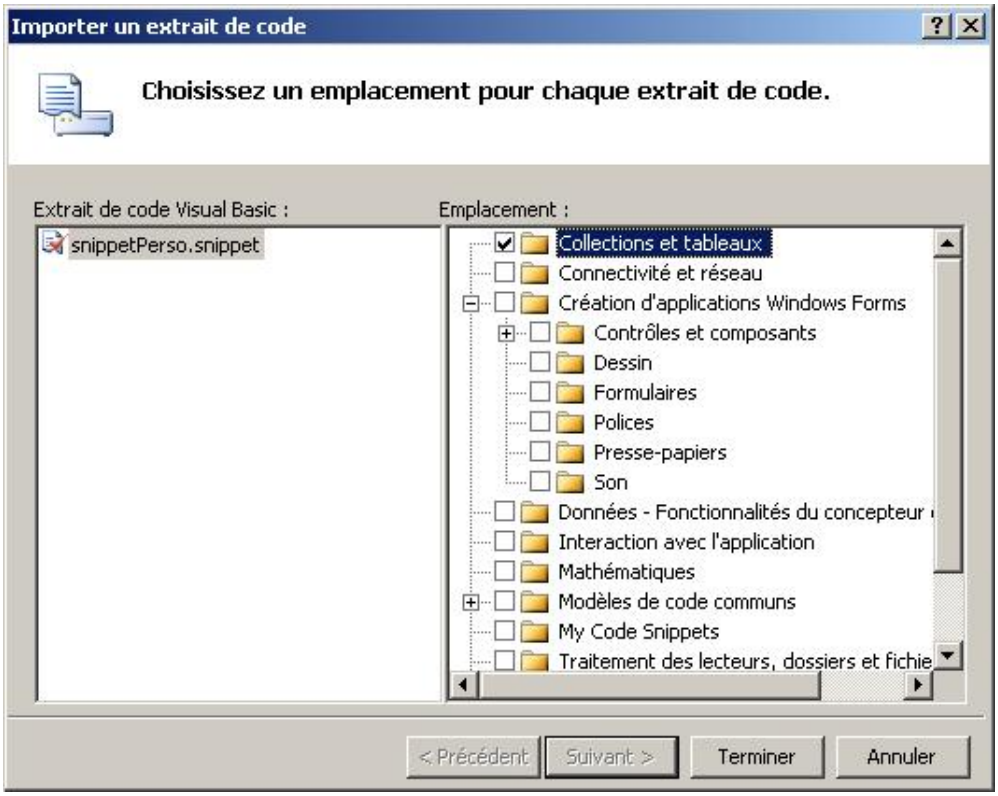
```
      <![CDATA[
      dim $nomTableau$($tailleTableau$) as $typeTableau$
      dim index as integer
      for index=0 to $taillaTableau$ -1
      ' inserer le code de traitement du tableau
      next ]]>
```

Vous enregistrez ensuite le fichier et votre Snippet est prêt. Il convient de maintenant l'intégrer dans Visual Studio. Pour cela, vous activez le gestionnaire de Snippet par le menu **Outils - Gestionnaire des extraits de code**.



Le bouton**Importer** permet d'ajouter votre Snippet à ceux déjà disponibles dans Visual Studio.

Après avoir sélectionné le fichier contenant le Snippet, vous devez choisir la rubrique dans laquelle il sera rangé.



Votre Snippet est maintenant disponible dans l'éditeur de code.

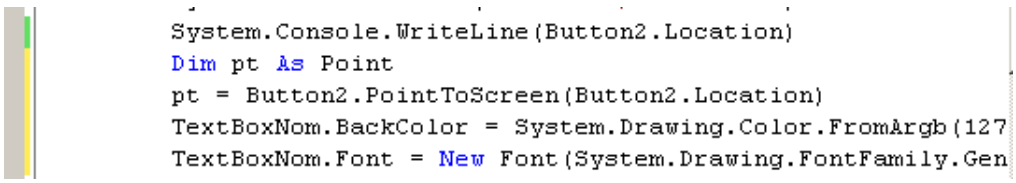
Il ne vous reste plus qu'à personnaliser le code généré.

```
Dim leTableau(tailleDuTableau) As Integer
Dim index As Integer
For index = 0 To tailleDuTableau - 1
    ' inserer le code de traitement du tableau
Next
```

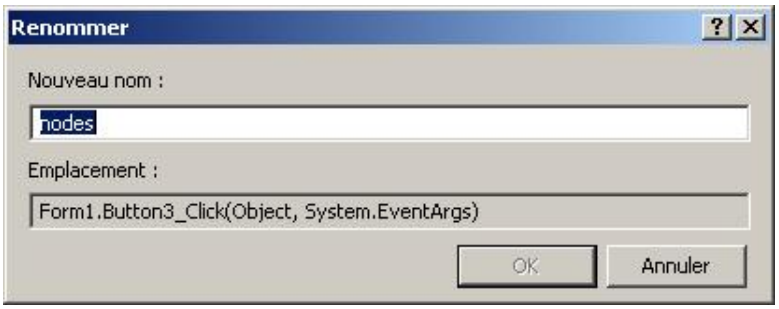
b. Suivi des modifications

Vous pouvez visualiser les portions de code ayant été modifiées depuis le démarrage de Visual Studio. Les modifications sont identifiées par une bordure de couleur apparaissant dans la marge de l'éditeur de code.

- Une bordure jaune indique que le code a été modifié mais pas encore sauvegardé.
- Une bordure verte indique que le code a été modifié et sauvegardé.



Vous pouvez aussi facilement renommer un élément et propager automatiquement la modification au reste du code. L'utilisation typique est le changement de nom d'une variable ou d'une classe. Vous ne devez pas renommer la variable directement dans le code mais utiliser la boîte de dialogue affichée en utilisant l'option **Renommer** du menu contextuel de l'éditeur de code sur le nom actuel de la variable.



La modification réalisée par l'intermédiaire de cette boîte de dialogue est répercutée sur l'ensemble du code où cette variable est utilisée.

c. Utilisation de macros

Comme la majorité des outils Microsoft, Visual Studio est maintenant capable de gérer les macros. Elles vous permettent de facilement enregistrer une série d'actions exécutées dans Visual Studio et de les reproduire par un simple clic sur un bouton d'une barre d'outils.

Nous allons créer trois macros permettant l'ajout d'instruction Imports pour les espaces de noms `System.Data.SqlClient`, `System.Data.OleDb` et `System.Data.Odbc`. Ces macros seront ensuite associées à trois boutons d'une nouvelle barre d'outils.

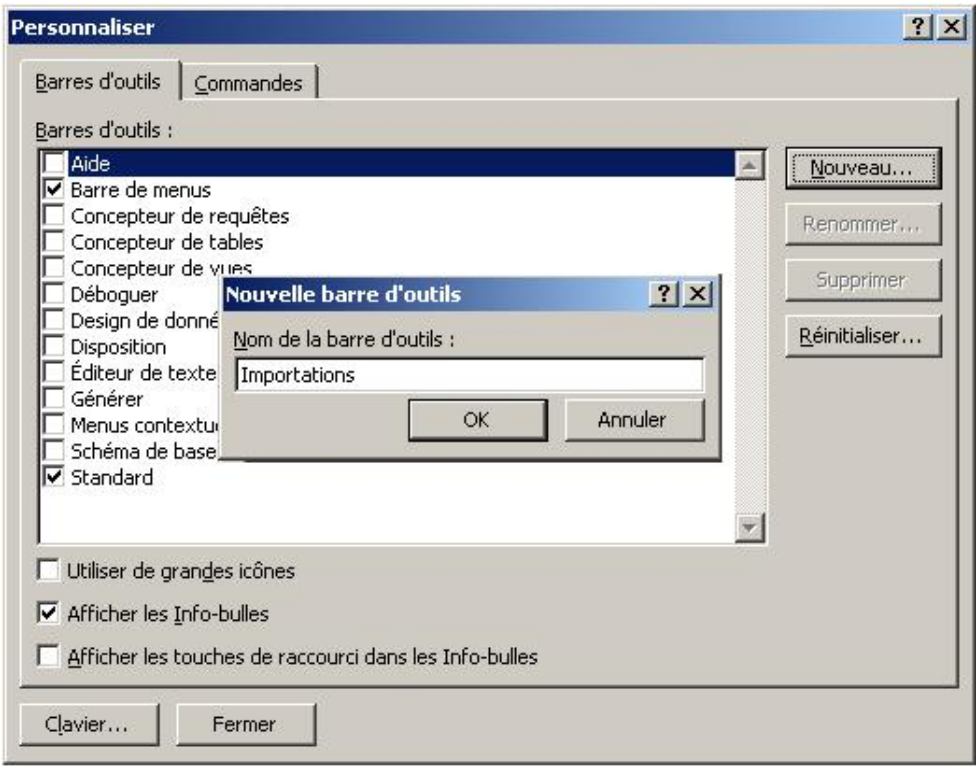
La première étape est d'enregistrer les macros comme on enregistre une séquence avec un magnétophone. Le menu **Outils - Macros - Enregistrer TemporaryMacro** déclenche l'enregistrement de vos actions. Vous pouvez alors saisir le code désiré, puis arrêter l'enregistrement grâce à la barre d'outils affichée au début de l'enregistrement de la macro.

Vous devez ensuite sauvegarder la macro par le menu **Outils - Macros - Sauvegarder TemporaryMacro**.

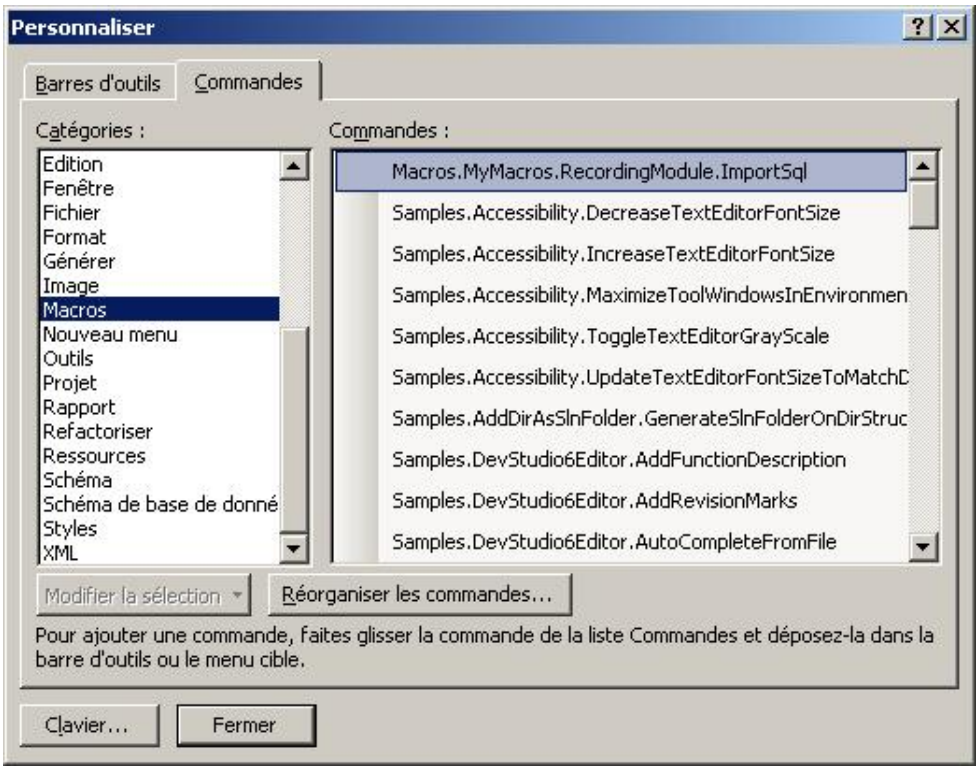
Effectuez ces opérations pour chacune des trois lignes de code suivantes en donnant un nom différent à chacune des macros.

```
Imports System.Data.SqlClient
Imports System.Data.OleDb
Imports System.Data.Odbc
```

Pour rendre plus facile l'utilisation de ces macros, nous allons les regrouper sur une nouvelle barre d'outils. Vous devez tout d'abord créer une nouvelle barre d'outils en utilisant l'option **Personnaliser** du menu contextuel disponible sur une barre d'outils existante.



La barre d'outils est maintenant disponible mais ne contient aucun bouton. Vous pouvez maintenant ajouter vos boutons à l'aide de la boîte de dialogue de personnalisation de la barre d'outils.



L'ajout des boutons se fait en effectuant un glisser-déplacer de la commande vers la barre d'outils. Les commandes apparaissent sous forme de texte sur la barre d'outils. Avant de fermer la boîte de dialogue de personnalisation, vous devez associer des images aux différentes commandes en utilisant l'option **Modifier l'image du bouton** du menu

contextuel de chaque commande.

Choisissez également pour chaque bouton le style par défaut. Votre barre d'outils est maintenant prête à être utilisée.

Maintenant que vous êtes familiarisé avec l'environnement de développement, vous devez vous familiariser avec le code Visual Basic.

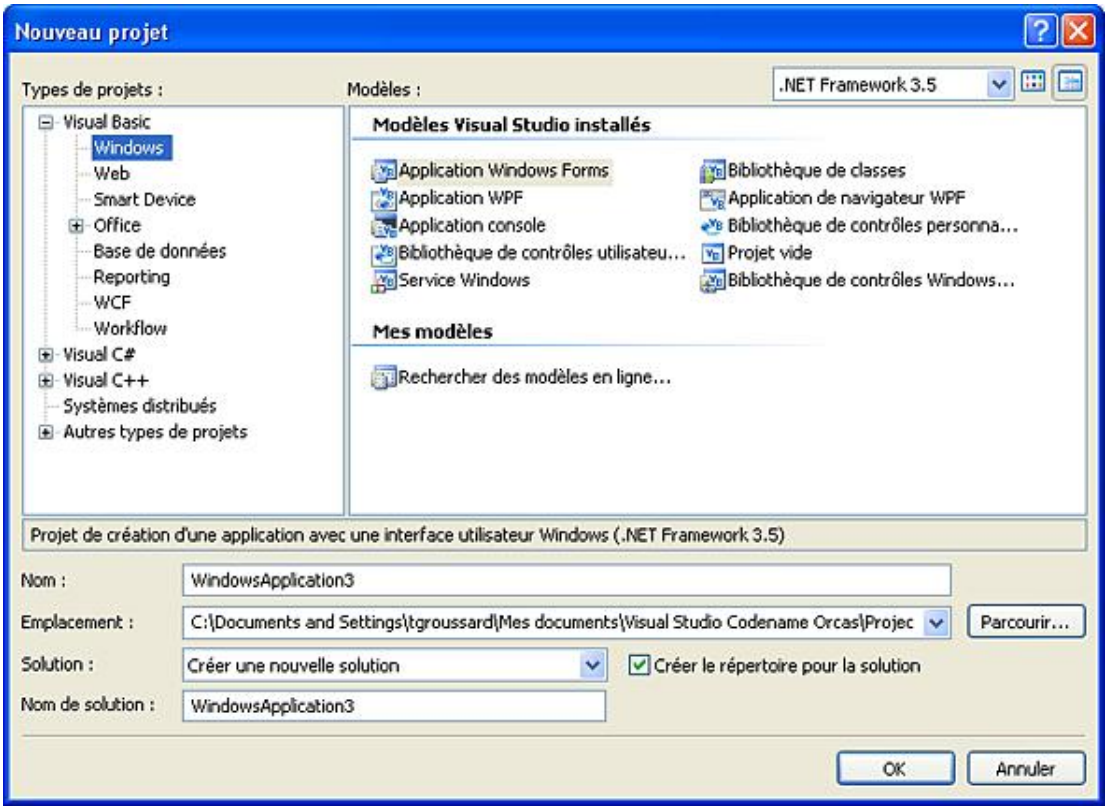
Les solutions

1. Présentation

Pour vous aider dans la création d'applications, Visual Studio vous propose plusieurs éléments servant à regrouper les composants d'une application. Le conteneur de plus haut niveau est la solution dans laquelle vous pourrez placer un ou plusieurs projets. Ces projets contiendront, à leur tour, tous les éléments pour que le compilateur soit capable de générer le fichier exécutable ou dll du projet. L'explorateur de solutions va nous permettre de manipuler tous ces éléments.

2. Création d'une solution

La création d'une solution est automatique lorsque vous démarrez un nouveau projet dans Visual Studio. Lors de la création du nouveau projet, il vous sera demandé plusieurs informations le concernant.



Par l'intermédiaire de cette boîte de dialogue, vous allez fournir les informations suivantes :

- la version du Framework nécessaire pour utiliser l'application,
- le langage utilisé pour développer le projet,
- le type de projet à créer,
- le nom du projet,
- le répertoire de base où seront stockés les fichiers,
- le nom de la solution,
- la création d'un répertoire pour la solution.

Après validation de cette boîte de dialogue, l'explorateur de solutions vous présente la nouvelle solution sur laquelle vous allez pouvoir travailler. Tous les fichiers de votre solution sont déjà créés et sauvegardés sur votre disque, à l'emplacement que vous avez spécifié.

Une solution contiendra au moins les fichiers suivants :

- Un fichier avec l'extension .sln, qui est le fichier de configuration de la solution. Ce fichier contient entre autres la liste de tous les projets composant la solution. Il est complété au fur et à mesure que vous ajoutez des nouveaux projets à la solution.
- Un fichier avec l'extension .suo, enregistrant les options associées à la solution. Ce fichier permet de retrouver ces options.
- Un fichier pour le projet, portant l'extension .vbproj. Ce fichier contient toutes les informa- tions de configuration du projet avec notamment la liste des fichiers constituant le projet, la liste de références utilisées par ce projet, les options à utiliser pour la compilation du projet, etc.
- De nombreux fichiers ayant l'extension .vb qui vont contenir le code source de toutes les classes, feuilles, modules constituant le projet.
- Un fichier .resx associé à chaque feuille de votre application. Ce fichier au format XML contient entre autres la liste des ressources utilisées sur cette feuille.
- Au final, une solution contient de nombreux autres fichiers en fonction des éléments utilisés dans votre projet (accès à une base de données, fichiers html...).

3. Modification d'une solution

Les solutions étant des conteneurs, il est bien sûr possible de gérer leurs éléments. Vous pouvez ajouter, supprimer, renommer des éléments dans la solution.

a. Ajouter un projet

Plusieurs possibilités sont disponibles pour l'ajout d'un projet :

- Si vous souhaitez créer un nouveau projet, choisissez l'option **Nouveau Projet** du menu **Fichier - Ajouter**. Une boîte de dialogue vous propose alors de configurer les caractéristiques du nouveau projet. Cette boîte de dialogue vous propose notamment un répertoire par défaut pour l'enregistrement du projet. Si ce répertoire ne correspond pas à l'emplacement où vous désirez enregistrer le projet, vous pouvez sélectionner un nouvel emplacement. Cette opération sera à réaliser pour chaque projet que vous ajouterez. Il peut être intéressant de modifier le chemin proposé par défaut pour l'enregistrement des projets. Pour cela, ouvrez le menu **Outils - Options**, puis dans la boîte de dialogue choisissez l'option **Projets et solutions** et modifiez la rubrique **Visual studio projects location**.
- Si vous souhaitez ajouter un projet déjà existant, vous devez utiliser l'option **Projet existant** du menu **Fichier - Ajouter**. Une boîte de dialogue de sélection de fichiers vous permet alors de choisir le fichier .vbproj du projet que vous souhaitez ajouter à la solution.

À noter que le projet reste à son emplacement d'origine sur le disque.

b. Supprimer un projet

- Pour supprimer un projet, utilisez le menu contextuel de l'explorateur de solutions en effectuant un clic droit sur le nom du projet que vous souhaitez supprimer de la solution.

Le projet est éliminé de la solution, mais reste enregistré sur le disque. Pour le supprimer définitivement, utilisez l'explorateur Windows pour supprimer les fichiers de ce projet. Si vous n'effacez pas les fichiers, le projet peut, par la suite, être de nouveau ajouté à une solution.

c. Renommer un projet

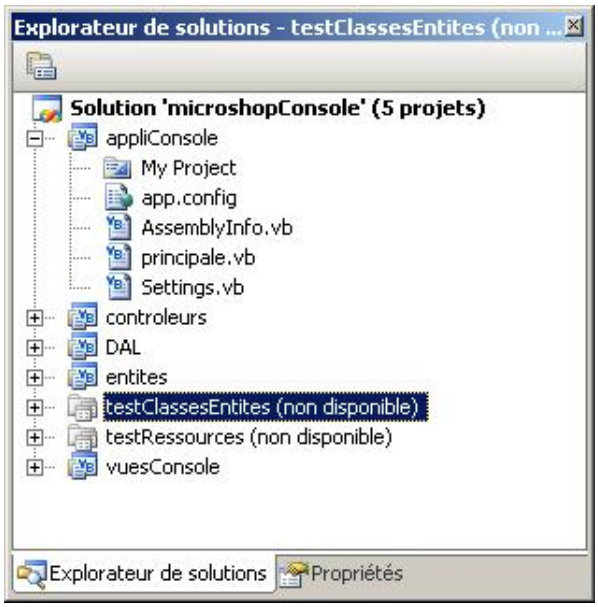
- Pour renommer un projet, utilisez le menu contextuel de l'explorateur de solutions en effectuant un clic droit sur le nom du projet que vous souhaitez renommer.

Le nom du projet devient alors modifiable dans l'explorateur de solutions. Cette modification concerne uniquement le nom du fichier .vbproj associé au projet. Elle ne modifie en aucun cas le nom du répertoire dans lequel se trouvent les fichiers du projet.

d. Décharger un projet

Lorsque vous souhaitez exclure temporairement un projet du processus de génération ou rendre l'édition de ses composants impossible, vous pouvez décharger le projet de la solution grâce à l'option **Décharger le projet**.

Un projet déchargé n'est pas retiré de la solution mais simplement marqué comme indisponible.



Le projet peut, bien sûr, être réhabilité dans la solution en utilisant l'option **Recharger le projet** du menu contextuel.

4. Organisation d'une solution

Si vous travaillez avec une solution contenant de très nombreux projets, vous pouvez ajouter un niveau de hiérarchisation en créant des dossiers de solutions. Ceux-ci permettent le regroupement logique de projets au sein d'une solution.

- Pour cela, créez, dans un premier temps, les dossiers dans la solution, puis organisez les projets dans ces dossiers.

Les dossiers de solutions ne créent pas de dossiers physiques sur disque, ils sont juste des conteneurs logiques à l'intérieur de la solution.

a. Création d'un dossier de solution

Un dossier de solution peut être créé par trois méthodes différentes.

- Pour toutes ces méthodes, sélectionnez la solution dans l'explorateur de solution.

- Ensuite, utilisez soit le bouton de la barre d'outils de l'explorateur de solution, soit le menu **Projet - Ajouter un nouveau dossier de solution** ou encore le menu contextuel disponible par un clic droit sur le nom de la solution.

Quelle que soit la méthode utilisée, vous devez fournir un nom pour le dossier créé.

b. Créer un projet dans un dossier

La création d'un projet dans un dossier de solution est identique à la création d'un projet directement dans la solution.

- Sélectionnez simplement, au préalable, le dossier dans lequel vous souhaitez créer le projet.

c. Déplacer un projet dans un dossier

Il arrive fréquemment que la nécessité d'organiser une solution avec des dossiers se fasse sentir alors que des projets existent déjà dans la solution.

- Créez, dans ce cas, les dossiers puis effectuez un glisser-déplacer des projets dans les dossiers correspondants.

5. Le dossier Éléments de solution

Les solutions contiennent principalement des projets, cependant il est possible d'avoir, dans une solution, des fichiers gérés indépendamment d'un projet particulier mais associés à la solution. C'est le cas, par exemple, d'un fichier icône que vous souhaitez utiliser dans plusieurs projets de la solution. Ces fichiers sont appelés éléments de solution et sont placés dans un dossier spécifique de la solution.

- Pour ajouter un nouvel élément de solution, utilisez le menu contextuel sur le nom de la solution en choisissant l'option **Ajouter - Nouvel élément** ou l'option **Ajouter - Élément existant**.

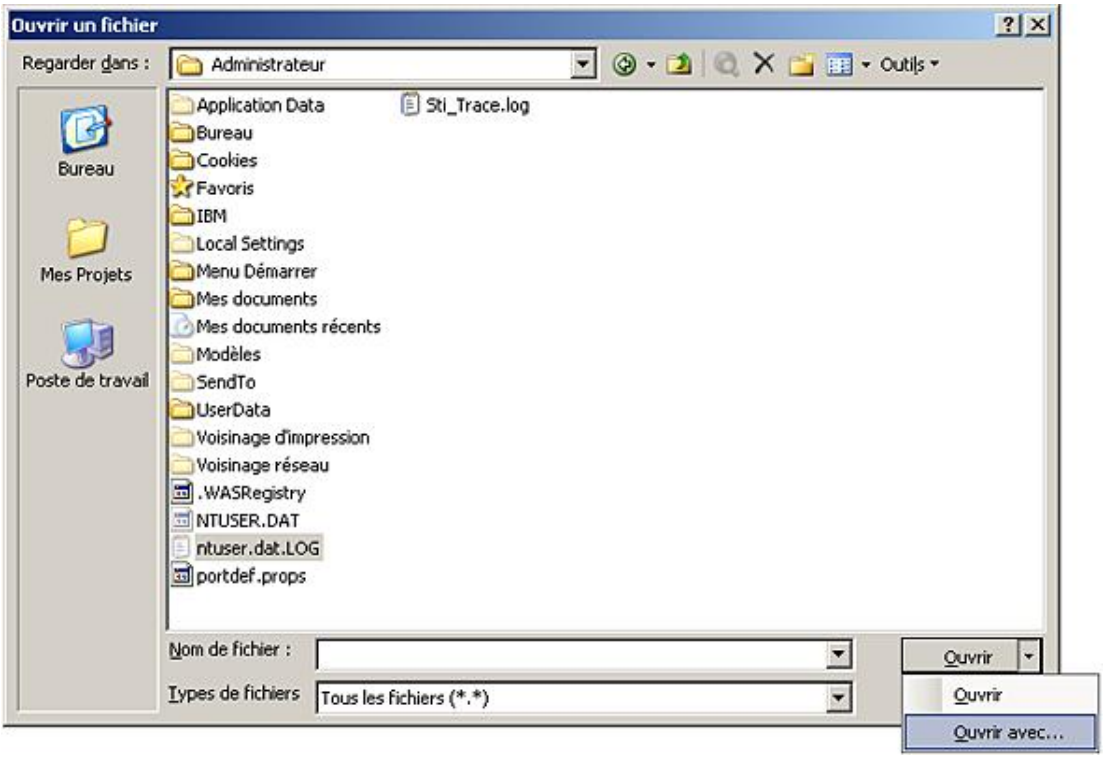
Le nouvel élément est alors ajouté dans le dossier **Éléments de solution**. Il est à noter que ce dossier n'existe pas, par défaut, dans une solution mais il est créé automatiquement lors de l'ajout du premier élément de solution. Les éléments de solution peuvent ensuite être modifiés avec un éditeur spécifique au type de fichier créé.

6. Le dossier Fichiers divers

Vous pouvez, parfois, vouloir visualiser le contenu d'un fichier alors que vous travaillez sur une solution, comme par exemple le compte rendu d'une réunion. Ce fichier ne doit pas faire partie de manière permanente de la solution. Vous pouvez l'ouvrir avec un éditeur externe et basculer entre Visual Studio et cet éditeur externe, mais il est plus pratique de visualiser le fichier directement dans l'environnement Visual Studio.

- Utilisez l'option **Ouvrir un fichier** du menu **Fichier**.

La boîte de dialogue vous permet de choisir le fichier à ouvrir. En fonction du type de fichier, un éditeur par défaut lui sera automatiquement associé pour permettre sa modification. Il est parfois utile de pouvoir choisir l'éditeur associé à un fichier. Pour cela, le bouton **Ouvrir** de la boîte de dialogue dispose d'un menu proposant l'option **Ouvrir avec** permettant le choix de l'éditeur associé au fichier.



La boîte de dialogue suivante vous propose la liste des éditeurs disponibles.

- Choisissez l'éditeur associé au fichier sur lequel vous souhaitez travailler et validez.

Le fichier est alors disponible dans le dossier Fichiers divers de la solution. Comme le dossier Eléments de solution, le dossier Fichiers divers n'existe pas, par défaut, dans la solution mais est créé automatiquement lors de l'ouverture d'un fichier.

Il sera visible dans l'explorateur de solutions uniquement si l'option correspondante est activée dans l'environnement Visual Studio. Pour cela, ouvrez le menu **Outils - Options**, puis dans la boîte de dialogue, choisissez l'option **Environnement - Documents** et activez l'option **Afficher les fichiers divers dans l'explorateur de solutions**. Comme le dossier Éléments de solution, ce dossier est un dossier "logique" et ne correspond pas à un emplacement sur le disque.

7. Configuration d'une solution

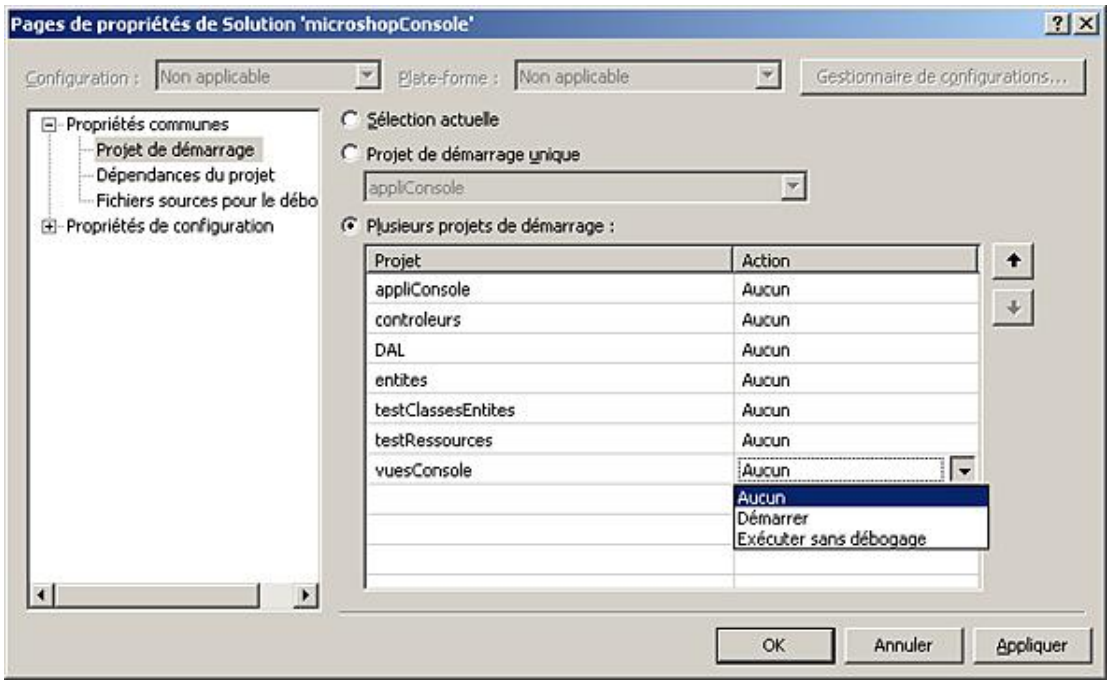
Les solutions disposent de propriétés permettant de configurer leurs comportement lors de la génération ou de l'exécution de l'application. Ces différentes propriétés sont regroupées dans une boîte de dialogue accessible par l'option **Propriétés** du menu contextuel d'une solution. Quatre catégories de propriétés sont disponibles :

- Projet de démarrage
- Dépendances de projets
- Fichiers sources pour débogage
- Configurations.

Regardons dans le détail chacune d'entres elles.

a. Configuration du projet de démarrage

Cette page de propriétés de la solution détermine, parmi les projets disponibles, celui ou ceux lancés au démarrage de la solution.



Deux options sont disponibles :

Projet de démarrage unique

Une combobox propose la liste des projets disponibles dans la solution parmi lesquels vous devez choisir celui qui sera exécuté au démarrage de la solution. Ce projet est par la suite signalé dans l'explorateur de solution par son nom apparaissant en gras. Cette sélection peut également se faire par le menu contextuel de l'explorateur de solutions en choisissant l'option **Définir comme projet de démarrage**.

Plusieurs projets de démarrage

Un tableau présente la liste de tous les projets disponibles dans la solution. Pour chacun d'eux, vous devez indiquer l'action à exécuter lors du lancement de l'application. Les choix possibles sont :

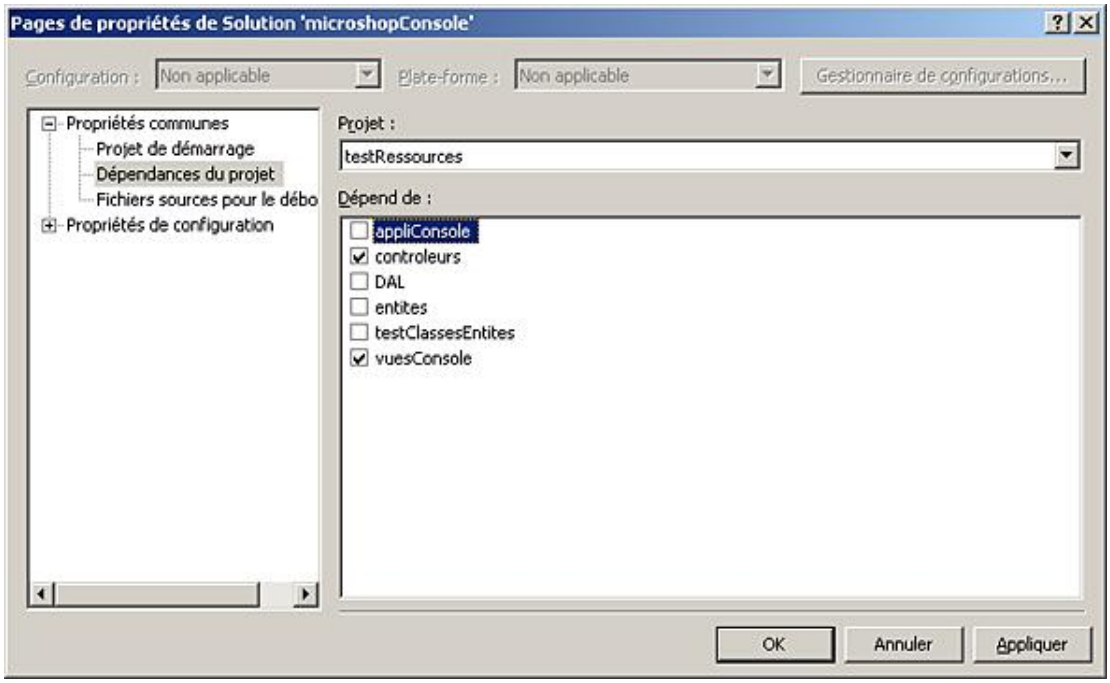
- Aucune action
- Démarrer le projet
- Exécuter le projet sans débogage.

Si vous choisissez de démarrer plusieurs projets au lancement de la solution, vous devez également indiquer l'ordre dans lequel ces projets seront démarrés. Cet ordre correspond en fait à l'ordre des projets dans le tableau. Les boutons  et  permettent de modifier cet ordre.

b. Dépendances de projet

La génération de certains projets nécessite au préalable la génération d'autres projets. C'est le cas, par exemple, si vous demandez la génération d'un projet qui utilise une référence vers un autre projet : celui-ci est alors une dépendance du projet initial.

La page de propriétés suivante permet de configurer ces dépendances.



- Dans la liste des projets, sélectionnez celui pour lequel vous souhaitez configurer les dépendances. Les autres projets de la solution sont alors listés, avec, pour chacun d'eux, une case à cocher. Lors de la génération du projet, tous les projets dont il dépend seront automatiquement régénérés, s'ils ont été modifiés depuis la dernière génération ou s'ils n'ont jamais été générés.

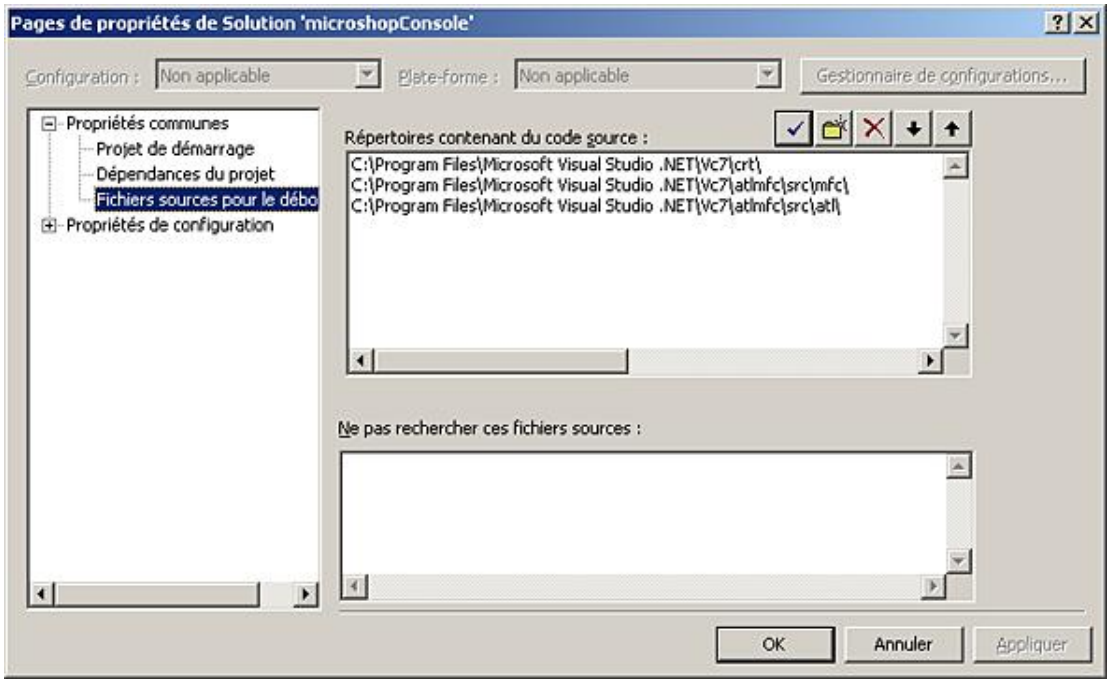
  Certaines dépendances ne peuvent être modifiées, la case à cocher apparaît alors en gris.

  C'est en général le cas, lorsqu'un projet possède une référence sur un autre projet ou que l'ajout d'une dépendance risque de créer une boucle. Par exemple, le projet1 dépend du projet2 et inversement.

Les dépendances de projet peuvent être également configurées par le menu contextuel de l'explorateur de solutions avec l'option **Dépendances du projet**.

c. Fichiers source pour le débogage

Lors du débogage d'une application, l'environnement Visual Studio a besoin d'accéder au fichier source du code qu'il est en train de déboguer. Cette page de propriété permet de spécifier les répertoires qui seront analysés à la recherche du code source.



La liste **Répertoires contenant du code source** affiche le nom des répertoires qui seront scrutés à la recherche de code source, pendant le débogage d'une application. Cette liste peut être gérée par la barre d'outils dont les boutons permettent de :

 Vérifier l'existence du répertoire.

 Ajouter un nouveau répertoire.

 Supprimer le répertoire sélectionné de la liste.

 Déplacer le répertoire vers le bas dans la liste.

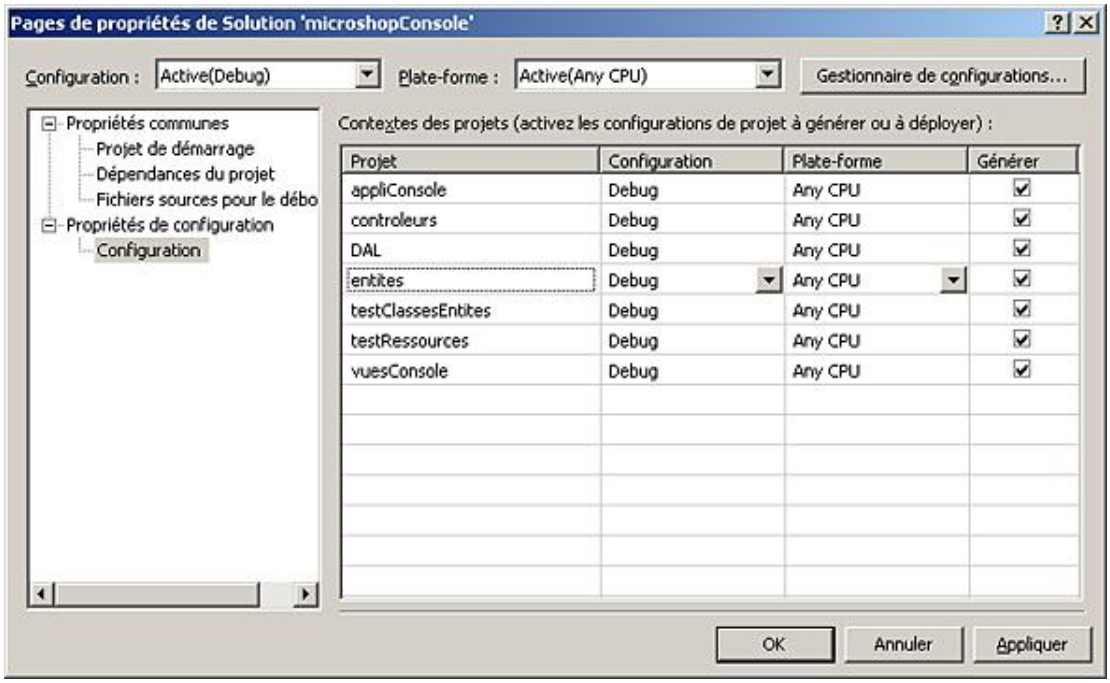
 Déplacer le répertoire vers le haut dans la liste.

La liste **Ne pas rechercher ces fichiers sources** exclut certains fichiers de la recherche.

d. Configurations

Les options de configuration permettent de définir comment différentes versions d'une solution et des projets qui la composent seront générées. Par défaut, deux configurations sont disponibles pour une solution dans Visual Studio : la configuration Debug et la configuration Release.

Pour chacun des projets présents dans la solution, les deux configurations seront également disponibles. Au niveau du projet, les configurations permettent de définir des options de compilations. La configuration Debug est utilisée pendant le développement et les tests du projet. La configuration Release est utilisée pour la génération finale du projet.



Nous avons en fait un système à trois niveaux : pour chaque configuration de solution, on indique quelle configuration utiliser pour chaque projet et, pour chaque configuration de projet, on spécifie des options de compilation. Les options de compilation sont modifiables au niveau des propriétés du projet.

Les projets

Les projets sont les conteneurs de deuxième niveau dans une application. Ils sont utilisés pour organiser logiquement, gérer, générer et déboguer les composants d'une application. La génération d'un projet produit, en général, un fichier exécutable ou une bibliothèque dll. Un projet peut être très simple et ne contenir que deux éléments, un fichier source (.vb) et le fichier de projet (.vbproj). Plus généralement, les projets contiennent de nombreux fichiers source, des scripts de base de données, des références vers des services Web, des ressources graphiques, etc.

Visual Studio propose par défaut un ensemble de modèles de projets. Ces modèles fournissent un point de départ pour la majorité des besoins dans le développement d'une application. Pour des cas plus spécifiques, vous pouvez créer vos propres modèles de projet.

1. Création d'un projet

- Pour la création d'un projet, activez le menu **Fichier - Nouveau projet**. Une boîte de dialogue vous propose alors de choisir les caractéristiques du nouveau projet.

- Choisissez tout d'abord la version du Framework pour laquelle vous souhaitez développer le projet. La version choisie influence les types de projets que vous pouvez créer.

- Choisissez ensuite le langage dans lequel vous souhaitez développer le projet. Les choix disponibles dépendent des langages installés dans Visual Studio. Dans notre cas, nous choisissons bien sûr Visual Basic.

- Choisissez ensuite le type de projet que vous souhaitez développer. La boîte de dialogue propose alors les différents modèles de projet disponibles en fonction du type de projet choisi.

- Après avoir fait votre choix, indiquez un nom pour le projet, un emplacement pour les fichiers du projet et un nom pour la solution. Le modèle sélectionné est alors utilisé par l'assistant pour créer les éléments du projet.

Après quelques instants le projet est disponible dans l'explorateur de solutions.

- Personnalisez maintenant l'ébauche créée.

a. Les modèles de projets

De nombreux modèles de projets sont disponibles dans Visual Studio. Ces modèles fournissent les éléments de base nécessaires pour développer chaque type de projet. Ils contiennent toujours au moins le fichier de projet, plus un exemplaire de l'élément le plus utilisé pour le type de projet correspondant ; par exemple, pour un projet de bibliothèque classe, un fichier source contenant une ébauche de classe est créé. Les modèles fournissent également des références et des importations par défaut pour les bibliothèques et les espaces de noms les plus utiles en fonction du type de projet.

Application Windows

Ce modèle de projet est certainement le plus utilisé. Il permet le développement d'application Windows standards. Le modèle ajoute au projet les éléments suivants :

- Un fichier AssemblyInfo.vb utilisé pour la description de l'application avec notamment les informations concernant la version.

- Un formulaire de base avec son fichier source form1.vb.

Les références suivantes sont automatiquement ajoutées et importées :

- System

- System.Core

- System.Data

- System.Data.DataSetExtensions
- System.Deployment
- System.Drawing
- System.Windows.Forms
- System.Xml
- System.Xml.Linq

Bibliothèque de classe

Ce modèle de projet est utilisable pour créer des classes et des composants qui pourront par la suite, être partagés avec d'autres projets. Les éléments suivants sont automatiquement ajoutés au projet :

- Un fichier AssemblyInfo.vb utilisé pour la description du projet avec notamment les informations concernant la version.
- Une classe de base avec son fichier source class1.vb.

Les références suivantes sont automatiquement ajoutées et importées :

- System
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Xml
- System.Xml.Linq

Bibliothèque de contrôles Windows

Comme le modèle précédent, ce type de projet permet de créer une bibliothèque de classes utilisable dans d'autres projets. Cette bibliothèque est plus spécifique, puisqu'elle est dédiée à la création de contrôles, utilisables par la suite dans une application Windows. Ces contrôles étendent la boîte à outils déjà disponible dans les applications Windows. Les éléments suivants sont automatiquement ajoutés au projet :

- Un fichier AssemblyInfo.vb utilisé pour la description du projet avec notamment les informations concernant la version.
- Une classe `UserControl1` héritant de la classe `System.Windows.Forms.User Control` fournissant les fonctionnalités de base pour un contrôle Windows, avec son fichier source `UserControl1.vb`.

Les références suivantes sont automatiquement ajoutées et importées :

- System
- System.Core
- System.Data

- System.Data.DataSetExtensions
- System.Drawing
- System.Windows.Forms
- System.Xml
- System.Xml.Linq

Application console

Ce type d'application est destiné à être exécuté à partir de la ligne de commande d'une fenêtre de invité de commande. Elle est bien sûr conçue sans interface graphique, les entrées/sorties se faisant à partir de la ligne de commande et vers la console.

Ce type d'application est très pratique pour réaliser des tests avec Visual Basic, car elle permet de se concentrer sur un point particulier sans avoir à se soucier de l'aspect présentation de l'application.

De nombreux exemples, présents dans cet ouvrage, sont basés sur une application en mode console. Il faut cependant avouer que, mis à part sa simplicité de création, ce type d'application est devenu un peu obsolète.

Les éléments suivants sont ajoutés par défaut au projet :

- Un fichier `AssemblyInfo.vb` utilisé pour la description du projet avec notamment les informations concernant la version.
- Une classe de base avec son fichier source `class1.vb`.

Les références suivantes sont automatiquement ajoutées et importées :

- System
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Deployment
- System.Xml
- System.Xml.Linq

Service Windows

Ce modèle de projet est conçu pour la création d'applications s'exécutant en tâche de fond sur le système. Le lancement de ce type peut être effectué automatiquement au démarrage du système et ne nécessite pas qu'une session utilisateur soit ouverte pour pouvoir s'exécuter.

Ce type d'application est dépourvu d'interface utilisateur. Si des informations doivent être communiquées à l'utilisateur, elles devront transiter par les journaux système, consultables, par l'observateur d'événements. Les éléments suivants seront ajoutés au projet :

- Un fichier `AssemblyInfo.vb` utilisé pour la description du projet avec notamment les informations concernant la version.
- Une classe de base avec le squelette des procédures `OnStart` et `OnStop` appelées automatiquement au démarrage et à l'arrêt du service.

Les références suivantes sont automatiquement ajoutées et importées :

- System
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Deployment
- System.ServiceProcess
- System.Xml
- System.Xml.Linq

Application WPF

Ce modèle de projet permet de bénéficier du nouveau système d'affichage graphique de Windows, utilisé dans Windows Vista.

Les éléments suivants sont automatiquement ajoutés au projet :

- Un fichier `AssemblyInfo.vb` utilisé pour la description de l'application avec notamment les informations concernant la version.
- Un fichier Application.Xaml et son fichier de code associé, Application.Xaml.vb, permettant la gestion des événements déclenchés au niveau de l'application.
- Une fenêtre de base Window1.Xaml et son fichier de code associé, Window1.Xaml.vb.

Les références suivantes sont automatiquement ajoutées et importées :

- PresentationCore
- PresentationFramework
- System
- System.Core
- System.Data
- System.Data.dataSetExtensions
- System.Xml
- System.Xml.Linq
- WindowsBase

Bibliothèque de contrôles WPF

Comme la bibliothèque de contrôles Windows, ce type de projet permet d'étendre la boîte à outils déjà disponible dans les applications WPF. Les éléments suivants sont ajoutés au projet.

- Un fichier `AssemblyInfo.vb` utilisé pour la description de l'application avec notamment les informations concernant la version.
- Un fichier UserControl1.xaml pour la définition de l'aspect graphique du contrôle.
- Un fichier UserControl1.xaml.vb pour le code associé à ce contrôle.

Les références suivantes sont automatiquement ajoutées et importées :

- PresentationCore
- PresentationFramework
- System
- System.Core
- System.Data
- System.Data.dataSetExtensions
- System.Xml
- System.Xml.Linq
- WindowsBase

Bibliothèque de contrôles WPF personnalisés

Ce type de projet a également pour vocation d'étendre la boîte à outils disponible pour les applications WPF. Contrairement au type de projet précédent, les contrôles ne sont pas créés de toute pièce, mais sont basés sur des contrôles existants dont ils étendent les caractéristiques.

Les références et importations sont identiques au type de projet précédent.

Projet vide

Ce modèle doit être utilisé lorsque vous souhaitez créer votre propre type de projet. Seul le fichier projet est créé. Par contre, aucun autre élément n'est ajouté automatiquement et aucune référence n'est créée ou importée.

b. Création de modèle de projet

Vous pouvez créer votre propre modèle de projet en fonction de vos habitudes de développement et faire en sorte qu'il apparaisse parmi les modèles prédéfinis.

Vous devez concevoir les éléments suivants :

- Un fichier de définition contenant les métadonnées du modèle. Ce fichier est utilisé par Visual Studio pour l'affichage du projet dans l'environnement de développement et pour l'affectation de propriétés par défaut au projet. Ces informations sont contenues dans un fichier XML ayant l'extension .vstemplate.
- Un fichier pour le projet (.vbproj).
- Les fichiers sources et ressources inclus par défaut lors de la création d'un projet à partir de ce modèle.

Ces fichiers doivent être compressés dans un fichier zip. Le fichier zip doit contenir les fichiers individuellement et non le dossier dans lequel ils sont placés.

Le fichier .vstemplate doit avoir le format suivant :

```
<VSTemplate Version="2.0.0"
xmlns="http://schemas.microsoft.com/developer/vstemplate/2005" Type="Project">
  <TemplateData>
    <Name>AppliPerso</Name>
    <Description>creation d'un projet avec une configuration personalise
</Description>
    <ProjectType>VisualBasic</ProjectType>
    <DefaultName>AppliPerso</DefaultName>
    </TemplateData>
  <TemplateContent>
    <Project File="AppliPerso.vbproj">
      <ProjectItem>AssemblyInfo.vb</ProjectItem>
      <ProjectItem>Feuille1.vb</ProjectItem>
      <ProjectItem>Feuille1.Designer.vb</ProjectItem>
      <ProjectItem>Feuille1.resx</ProjectItem>
    </Project>
  </TemplateContent>
</VSTemplate>
```

On retrouve dans ce fichier :

Dans la section Name

Le nom affiché par la boîte de dialogue de création d'un nouveau projet.

Dans la section Description

Une description détaillée du projet.

Dans la section Project Type

Le nom du dossier dans lequel ce projet sera classé dans la boîte de dialogue de création de projet.

Dans la section Default Name

Le nom utilisé par défaut pour tous les projets créés à partir de ce modèle. Ce nom est complété par un suffixe numérique à la création du projet.

Dans la section Project File

Le nom du fichier projet associé au modèle. Ce fichier doit être présent dans le fichier zip du modèle.

Dans les sections ProjectItem

Les éléments faisant partie du projet. Ces éléments doivent également être disponibles dans le fichier zip.

c. Modification d'un modèle existant

La modification d'un modèle consiste à utiliser un fichier zip existant contenant les éléments nécessaires au projet et y ajouter des éléments supplémentaires. Si des fichiers sont ajoutés au modèle, ils doivent être placés dans le fichier zip et également référencés dans le fichier .vstemplate. Les modèles prédéfinis de Visual Studio sont placés dans le répertoire C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\ProjectTemplates\ VisualBasic. Pour que les modifications soient prises en compte, vous devez mettre à jour le cache utilisé par Visual Studio. Pour cela :

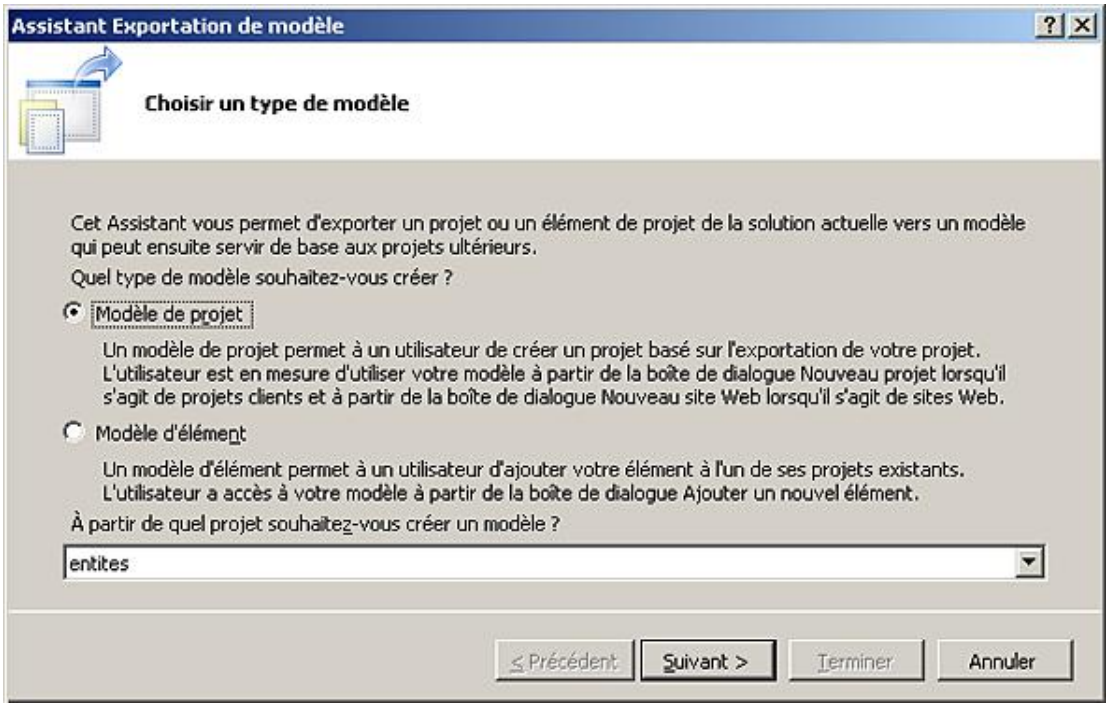
- Ouvrez une fenêtre de commande Visual Studio.
- Saisissez la commande `devenv /setup`. Soyez patient car cette commande est assez longue à s'exécuter. Après exécution de la commande, vos modifications sont disponibles dans le modèle de projet.

d. Utilisation d'un projet existant comme modèle

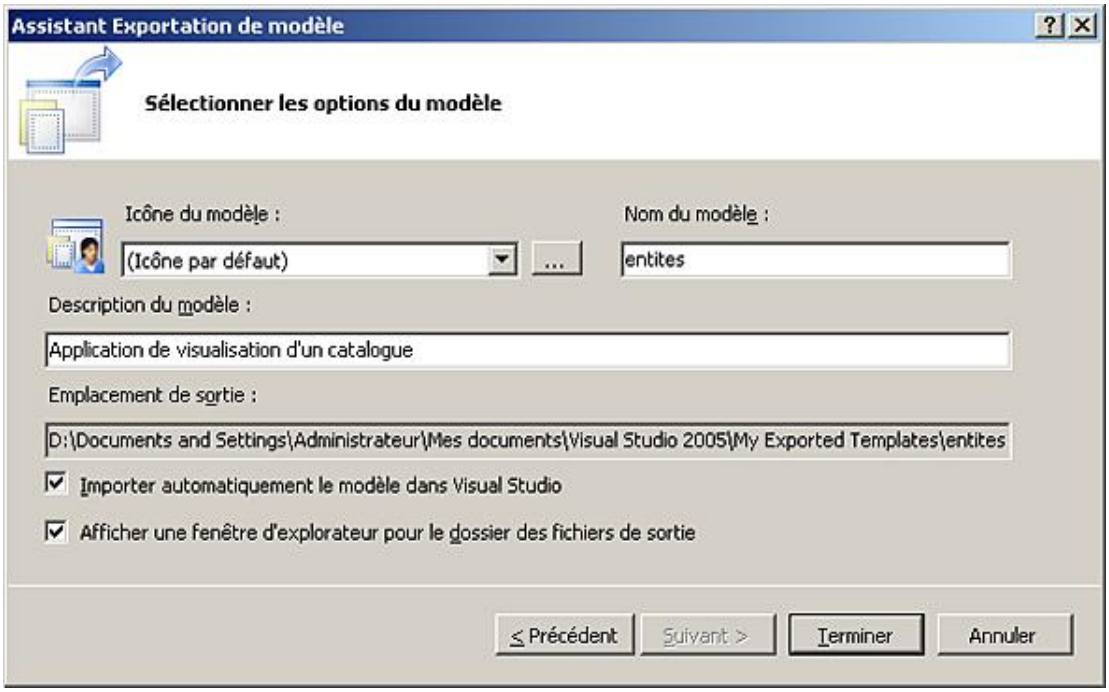
C'est peut-être la solution la plus simple pour construire un modèle de projet.

- Dans un premier temps, créez le modèle comme un projet ordinaire.

- Une fois votre projet finalisé, exportez-le comme modèle. Le menu **Fichier - Exporter le modèle** démarre un assistant pour vous guider pendant la création du modèle.



Cette première boîte de dialogue vous propose de choisir le projet que vous souhaitez exporter ainsi que, la rubrique de la boîte de dialogue de création de projet dans laquelle sera placé le futur modèle.



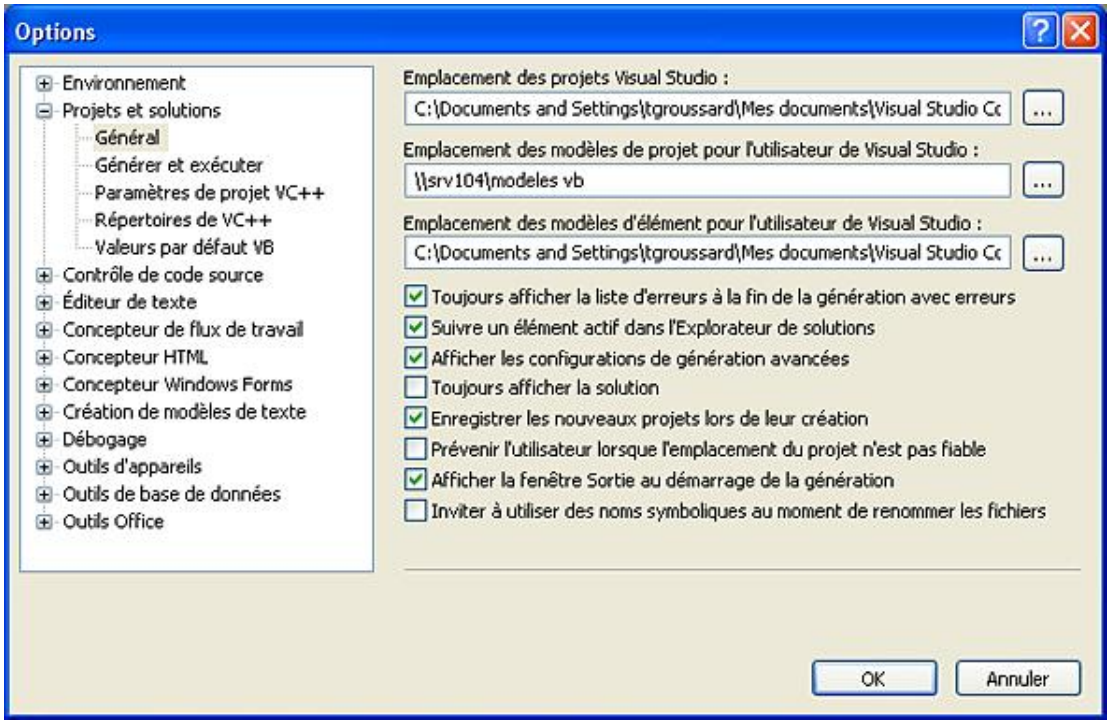
Cette deuxième boîte de dialogue vous invite à choisir une icône pour votre modèle de projet, un nom pour le modèle et une description. Deux options supplémentaires vous permettent de prendre en compte immédiatement le nouveau modèle dans Visual Studio et de vous présenter le résultat de la génération en vous affichant le contenu du fichier zip créé. Après validation de cette dernière boîte de dialogue, le nouveau modèle de projet est disponible dans Visual Studio.

Cette méthode est très simple pour construire un nouveau modèle de projet et évite de se torturer l'esprit avec la syntaxe du fichier .vstemplate.

Dans le cadre d'un développement en équipe, il peut être intéressant de partager les modèles personnalisés entre tous les membres de l'équipe.

- Recopiez les fichiers zip sur un partage réseau.

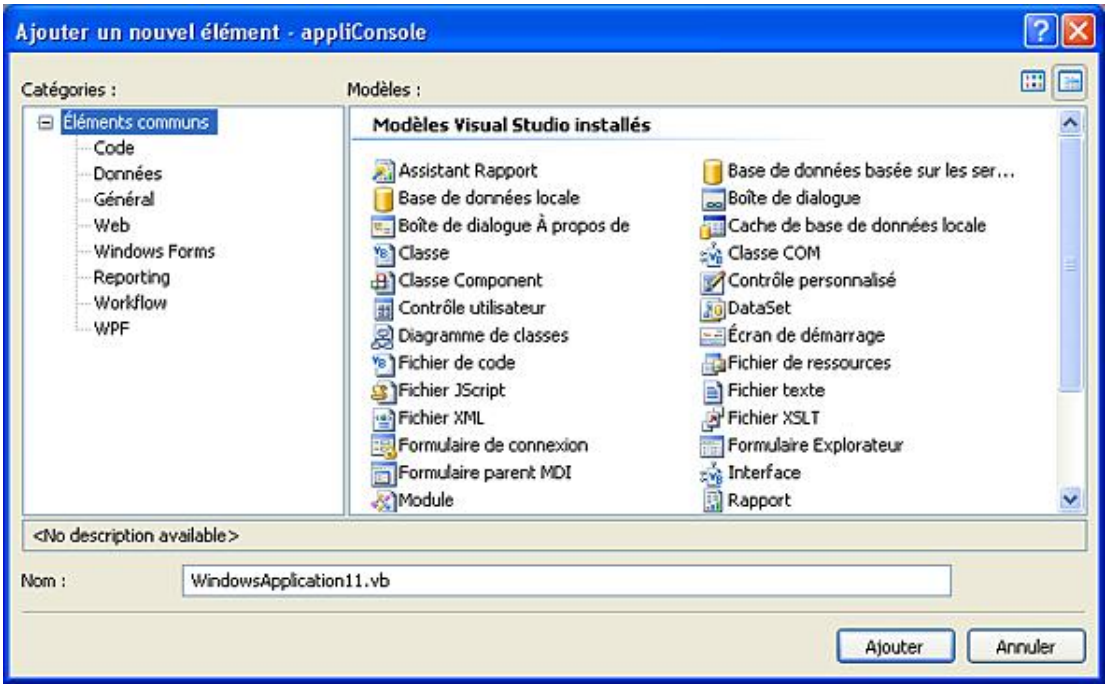
- Configurez l'environnement Visual Studio pour lui permettre d'accéder aux modèles. Cette modification s'effectue par la boîte de dialogue disponible par le menu **Outils - Options**.



2. Modification d'un projet

Les modèles de projets sont très utiles pour créer rapidement les bases d'une application mais, très fréquemment, nécessiteront l'ajout de nouveaux éléments au projet. Ces ajouts se font par l'intermédiaire du menu contextuel de l'explorateur de projet.

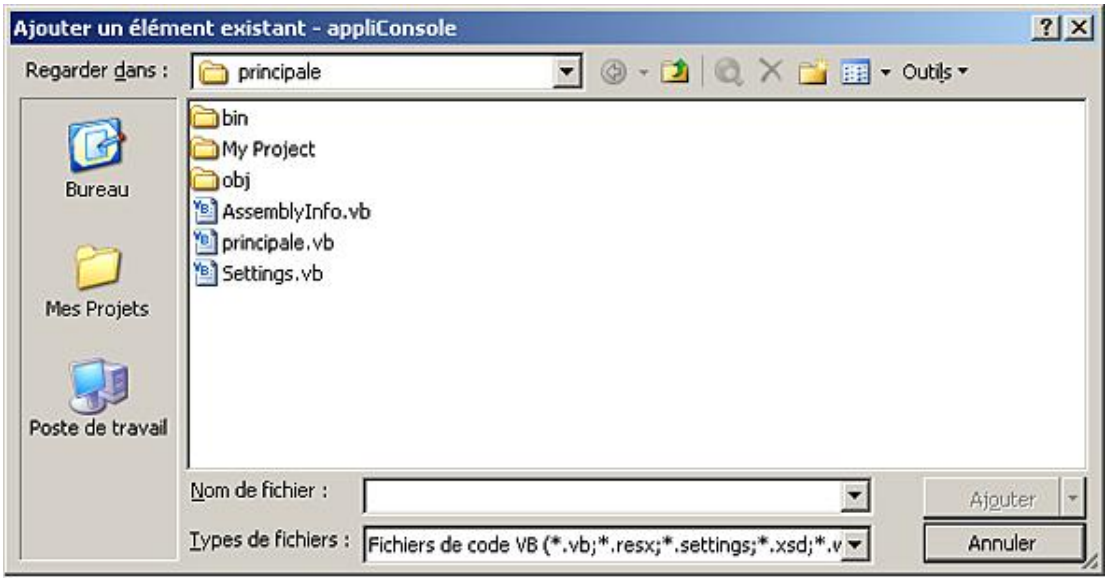
- Activez l'option **Ajouter - Nouvel élément** afin de choisir le type d'élément que vous souhaitez ajouter au projet. La boîte de dialogue propose un nombre impressionnant d'éléments pouvant être ajoutés à un projet.



- Indiquez ensuite un nom pour le fichier contenant le nouvel élément.

En fonction des types de projet, des options supplémentaires permettant rapidement d'ajouter un nouvel élément sont disponibles dans le menu contextuel. Celles-ci affichent simplement la boîte de dialogue précédente avec le type d'élément correspondant déjà présélectionné.

Il est également possible de reprendre un élément existant dans un autre projet et de l'ajouter à un projet. Utilisez, dans ce cas, l'option **Ajouter - Élement existant** du menu contextuel de l'explorateur de projets. Une boîte de dialogue vous propose la sélection du fichier à inclure dans le projet.



Le bouton **Ajouter** de cette boîte de dialogue comporte un menu permettant d'ajouter le fichier normalement (une copie locale du fichier est alors réalisée) ou de créer un lien sur le fichier (le fichier original est utilisé). Il faut être prudent avec cette possibilité car le fichier "n'appartient pas" réellement à l'application mais peut être partagé entre plusieurs applications. Si le fichier est supprimé du disque, toutes les applications l'utilisant ne pourront plus être compilées.

La gestion des fichiers dans l'explorateur de solutions est identique à la gestion des fichiers dans l'explorateur Windows. Les fichiers peuvent être copiés et collés ou déplacés par un cliqué-glissé d'un dossier à un autre. L'utilisation des touches [Ctrl], [Shift] et [Ctrl] [Shift] pendant le cliqué-glissé modifie l'action réalisée. Un cliqué-glissé au sein d'un même projet effectue un déplacement de fichier. S'il est réalisé entre deux projets, c'est alors une copie

de fichier qui est effectuée. Ce comportement peut être modifié par l'utilisation de la touche [Shift] lors du cliqué-glissé. Pour réaliser une copie de fichier au sein d'un projet, la touche [Ctrl] sera utilisée conjointement avec le cliqué-glissé. La création d'un lien s'effectue avec la combinaison de touches [Ctrl] [Shift] lors du cliqué-glissé.

Afin de retirer un élément d'un projet, deux options sont accessibles par le menu contextuel de l'explorateur de solutions :

- L'option **Supprimer** supprime le fichier du projet mais également du disque.
- L'option **Exclure du projet** retire le fichier du projet, mais ne le supprime pas du disque. Cette option est utile si d'autres projets utilisent ce fichier par l'intermédiaire d'un lien.

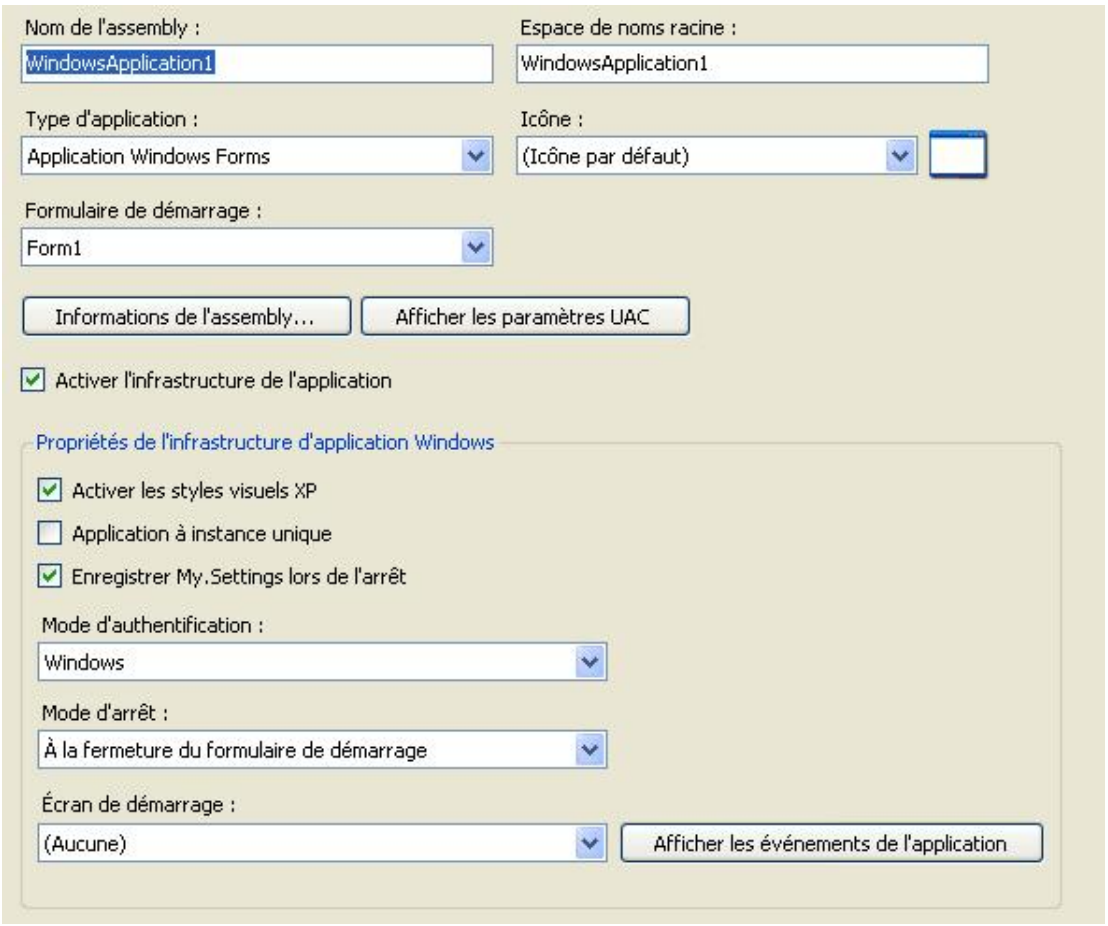
3. Propriétés des projets

Les projets sont des éléments fondamentaux de la conception d'une application avec Visual Basic. Ils possèdent de nombreuses propriétés permettant de modifier leurs comportements au moment de la conception ou de l'exécution de l'application. L'ensemble des propriétés sont accessibles par une boîte de dialogue présentant, par l'intermédiaire d'onglets, les différentes rubriques de configuration d'un projet.

- Activez cette boîte de dialogue par l'option **Propriétés** du menu contextuel de l'explorateur de projet ou par le bouton  de la barre d'outils de l'explorateur de projet.

a. Propriétés d'application

Les propriétés présentes sur cet onglet vont permettre de configurer le comportement de l'application.



Nom de l'assembly

Cette propriété détermine le nom utilisé pour le fichier résultant de la compilation de l'application. Par défaut, ce

fichier porte le même nom que le projet mais ils peuvent être modifiés indépendamment l'un de l'autre. L'extension associée au fichier dépend du type du projet.

Type d'application

Cette propriété détermine le type d'application générée par la compilation du projet. En règle générale, cette propriété est déterminée par le modèle choisi au moment de la création du projet. Cette propriété est très rarement modifiée par la suite car elle dépend énormément du code de votre projet (si vous avez conçu votre application comme une application Windows et que souhaitez la considérer comme une application console, il risque d'y avoir beaucoup de code inutile !).

Formulaire de démarrage

Cette propriété détermine le point d'entrée dans l'application, lors de son exécution. Généralement, elle correspond à la fenêtre principale de l'application ou à une procédure `Sub Main`. Cette propriété n'est disponible que pour les projets pouvant s'exécuter de manière autonome. Elle est inutile pour les projets de type bibliothèque.

Espace de noms racine

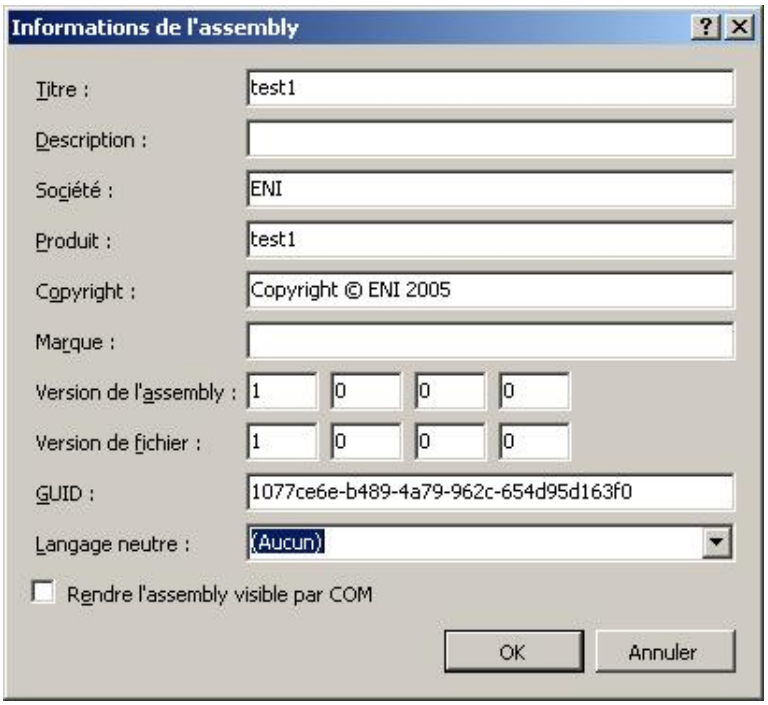
Tous les éléments du projet, accessibles à partir d'un autre projet, appartiennent à l'espace de nom défini par cette propriété. Celle-ci vient s'ajouter aux éventuels espaces de noms, définis au niveau du code lui-même. Par défaut, cette propriété correspond au nom du projet mais elle peut être modifiée indépendamment de celui-ci. Elle peut même être vide vous permettant ainsi de gérer les espaces de noms directement dans le code.

Icône

Cette propriété configure l'icône associée au fichier compilé du projet, lorsqu'il est affiché dans l'explorateur Windows ou lorsque l'application apparaît sur la barre des tâches de Windows.

Informations de l'assembly

Cette option permet de fournir des informations sur le code généré par la compilation du projet. Une boîte de dialogue permet de remplir différentes rubriques concernant la description du projet.



L'utilisateur de votre code pourra consulter ses informations en affichant les propriétés du fichier compilé dans l'explorateur Windows.

Afficher les paramètres UAC

Cette option permet de déterminer le niveau d'exécution requis pour l'application. Ces informations sont utilisées par le mécanisme *User Account Control* (UAC) de Windows Vista. Il détermine sous quelle identité va être exécuté le code de l'application. Trois valeurs sont possibles

- asInvoker : l'application s'exécute avec l'identité actuelle de l'utilisateur et ne demande pas d'augmentation

de privilèges.

- highestAvailable : l'application s'exécute avec le plus haut niveau de privilèges de l'utilisateur.
- requireAdministrator : l'application doit s'exécuter avec le privilège administrateur et UAC peut vous demander votre consentement pour accorder l'augmentation des privilèges.

Pour les systèmes autres que Windows Vista cette option est ignorée.

Activer l'infrastructure de l'application

Cette option détermine si vous souhaitez activer une interaction plus évoluée entre l'application et le système d'exploitation. Si cette option est active, l'élément de démarrage de l'application doit obligatoirement être une feuille. L'utilisation de cette option active la disponibilité des propriétés suivantes.

Activer les styles visuels XP

Si cette option est activée et que l'application s'exécute sur un système Windows XP, alors l'interface utilisateur de l'application s'adaptera au thème Windows actif.

Application à instance unique

Par défaut, vous pouvez lancer autant d'exemplaires d'une même application que vous le souhaitez sur un poste ; toutefois, il peut parfois être utile de n'autoriser le fonctionnement que d'un seul exemplaire de l'application à un instant donné (problème de licence d'utilisation, préservation des ressources de la machine...). L'activation de cette option garantit qu'il n'y aura pas plus d'un exemplaire de l'application s'exécutant sur la machine. Si une nouvelle instance est lancée alors qu'il en existe déjà une sur le système, le focus passe immédiatement sur l'instance existante. Au niveau de l'application, l'événement **StartupNextInstance** est également déclenché.

Enregistrer My.settings lors de l'arrêt

Cette option indique si les propriétés personnalisées de l'application sont sauvegardées à la fermeture de l'application. Ceci permet, par exemple, de mémoriser les préférences de l'utilisateur.

Mode d'authentification

Par défaut, les applications Visual Basic utilisent l'authentification Windows pour identifier l'utilisateur de l'application. Si vous souhaitez gérer vous-même cette identification, vous devez utiliser l'option **Défini au niveau de l'application**.

Mode d'arrêt

Cette option détermine le comportement de l'application lors de son arrêt. Par défaut, l'exécution de l'application s'arrête lorsque la fenêtre de démarrage de l'application est fermée, même s'il y a d'autres fenêtres actives (hormis bien sûr une fenêtre modale). L'option **À la fermeture du dernier formulaire** provoque l'arrêt de l'application à la fermeture de la dernière fenêtre active de l'application ou lorsque les instructions `My.application.exit` ou `end` sont appelées explicitement dans le code.

Écran de démarrage

Les écrans d'accueil sont souvent utilisés pour fournir des informations à l'utilisateur, pendant le démarrage de l'application. Visual Studio propose un modèle d'écran d'accueil personnalisable. Cet écran apparaît pendant le chargement de la fenêtre principale de l'application.

Afficher les événements de l'application

Cette option permet d'accéder aux gestionnaires d'événements de l'objet application. Ces gestionnaires d'événements permettent de réagir face à différentes situations :

Startup

L'application démarre.

StartUpNextInstance

Un nouvel exemplaire de l'application vient d'être lancé.

Shutdown

L'application s'arrête.

UnHandledException

Une exception non gérée vient de se produire.

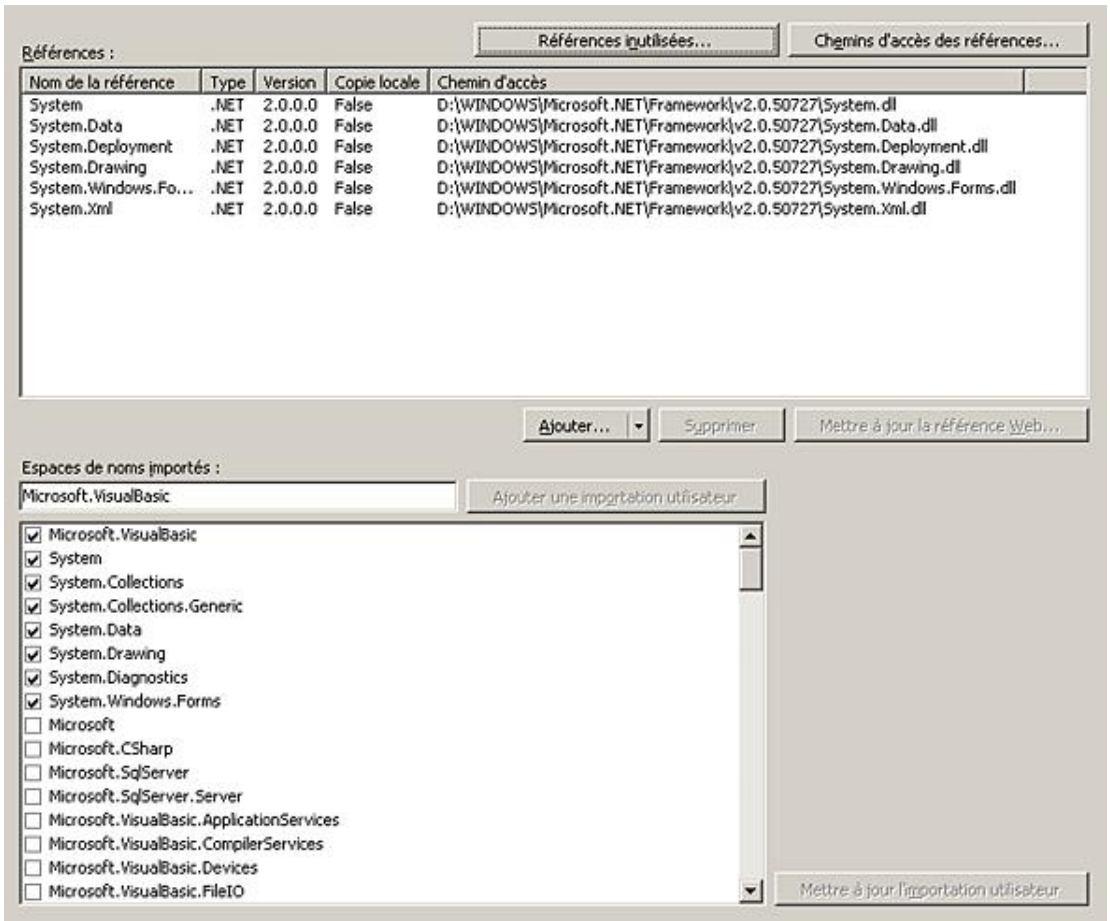
NetworkAvailabilityChanged

L'accès au réseau vient d'être modifié (le câble est débranché ou rebranché).

b. Références et importations d'un projet

Pour pouvoir utiliser des éléments externes, disponibles dans un assembly, vous devez ajouter une référence à cet assemblage.

La page de propriété suivante regroupe toutes les informations concernant les références d'un projet.



Dans cette page de propriété, la liste **References** présente tous les assemblages actuellement référencés par le projet.

Cette liste peut être mise à jour à l'aide des boutons **Ajouter**, **Supprimer**, **Mettre à jour la référence Web** qui permettent respectivement, d'ajouter une référence locale ou une référence Web, de supprimer une référence ou de mettre à jour une référence Web.

Le bouton **Chemins d'accès des références** permet d'indiquer des répertoires supplémentaires contenant des assemblages disponibles. Ces répertoires sont scrutés à l'ouverture de la boîte de dialogue d'ajout de référence et les éventuels assemblages qu'ils contiennent sont ajoutés à la liste des assemblages disponibles. Lorsqu'une référence est ajoutée à un assemblage, le fichier original est utilisé lors de la première exécution de l'application. Vous pouvez utiliser la création automatique d'une copie locale de ce fichier dans le répertoire de l'application. Pour cela :

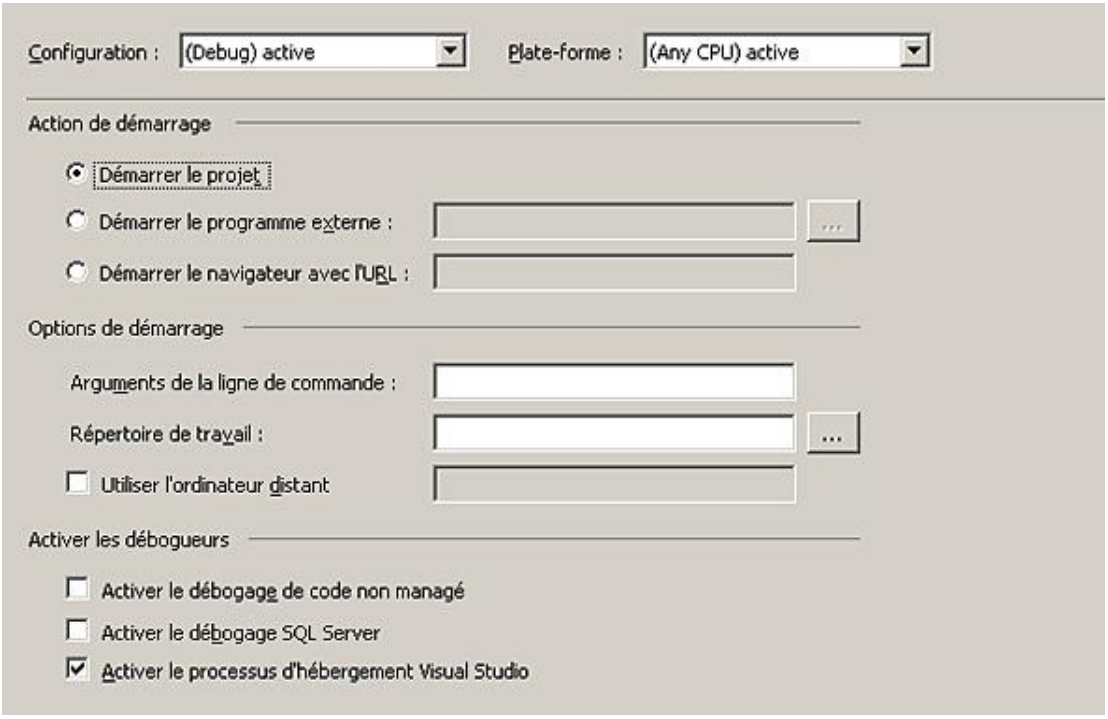
- Modifiez la propriété `Copy Local` de la référence concernée. Après avoir sélectionné la référence, vous pouvez modifier cette propriété dans la fenêtre de propriétés de Visual Studio.

Le bouton **Références inutilisées** propose la liste de toutes les références non utilisées dans le code et vous propose de les supprimer du projet.

Les éléments disponibles dans les assemblages référencés font très certainement partie d'un espace de nom. Pour pouvoir les utiliser facilement il est possible d'importer automatiquement certains espaces de nom. La liste **Espaces de noms importés** reprend les espaces de nom importés automatiquement dans tous les codes du projet. Vous pouvez compléter cette liste en saisissant dans la zone de texte le nom de l'espace de nom importé et en validant avec le bouton **Ajouter une importation utilisateur**.

c. Propriétés de débogage

Les propriétés présentes sur cette page déterminent le comportement du projet lors de son débogage.



Action de démarrage

Cette propriété détermine le comportement du projet lors du démarrage du débogage. Trois options sont possibles :

- **Démarrer le projet** indique que le projet lui-même doit être exécuté. Cette option n'est à utiliser que pour les projets d'application Windows ou les projets d'application console.
- **Démarrer le programme externe** permet de provoquer l'exécution d'une application externe qui va se charger de faire des appels au code notre projet. Cette option est utilisée pour le débogage de bibliothèques de classes.
- **Démarrer le navigateur avec l'URL** est identique à l'option précédente, mis à part que l'application démarrée est une application Web.

Options de démarrage

Arguments de la ligne de commande précise les arguments passés à l'application lors de son exécution par Visual Studio. Ces arguments peuvent être utilisés par le code pour déterminer l'action à entreprendre : par exemple, démarrer l'application en mode maintenance.

Répertoire de travail permet de spécifier le répertoire actif pendant l'exécution de l'application.

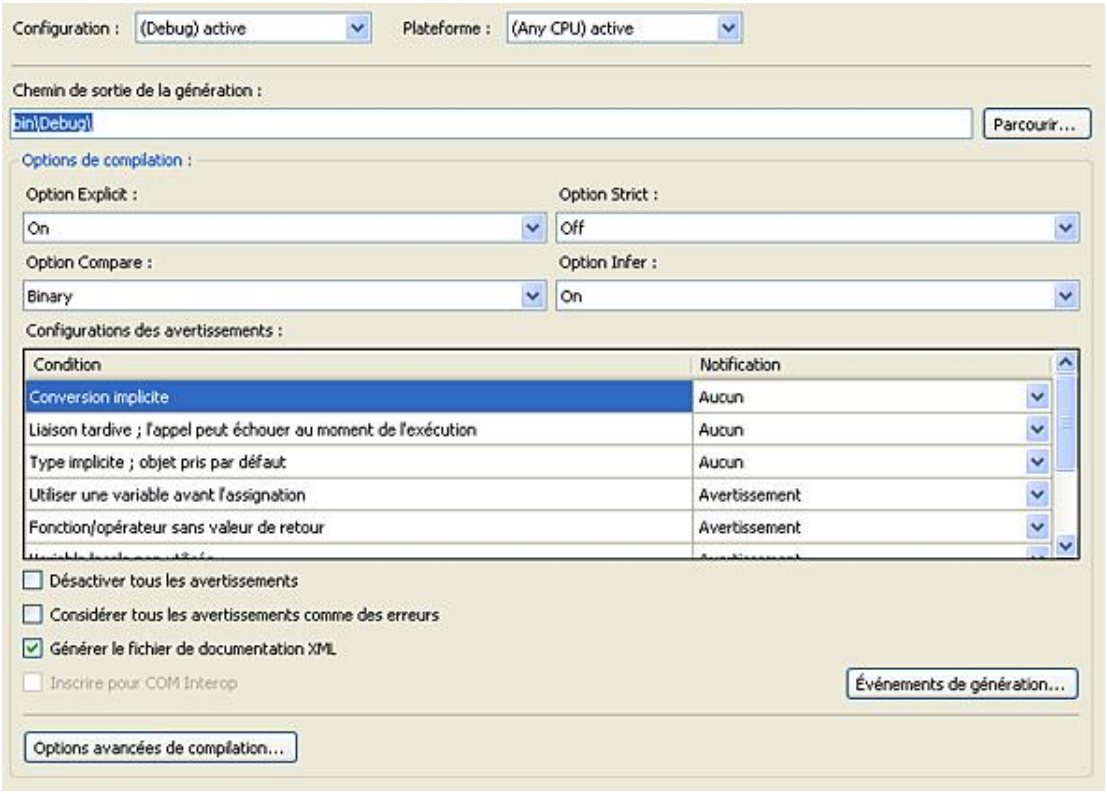
Utiliser l'ordinateur distant autorise le débogage d'une application s'exécutant sur une autre machine. Dans ce cas, le nom de la machine distante, sur laquelle le code va s'exécuter, est à indiquer.

Activer les débogueurs

Ces options déterminent les différents types de débogueur actifs, en complément du débogueur de code managé de Visual Studio.

d. Propriétés de compilation

Les propriétés de cette page concernent le fonctionnement du compilateur et son éventuelle optimisation.



Chemin de sortie de la génération

Cette propriété indique le répertoire dans lequel est copié le fichier résultant de la compilation du projet. Par défaut, il s'agit du sous-répertoire **bin** du répertoire dans lequel se trouve le projet.

Option Explicit

Cette option permet d'exiger ou non que toute variable soit déclarée avant d'être utilisée. Elle s'applique à tous les fichiers source d'un projet. Il est cependant possible, pour un fichier particulier, de modifier cette option en ajoutant la directive `Option Explicit On` ou `Option Explicit Off` en début de fichier.

Option Strict

Cette option permet de contrôler les conversions effectuées. Si une conversion restrictive est tentée, le compilateur génère une erreur. Comme l'option précédente, celle-ci s'applique à tous les fichiers source d'un projet et peut être modifiée pour un projet particulier avec la directive `Option Strict On` ou `Option Strict Off`.

Option Compare

Cette option détermine comment la comparaison de chaînes de caractères est effectuée dans l'application. Avec la valeur **Binary**, l'application fait une distinction entre les caractères en minuscule et en majuscule, lors d'une comparaison. La valeur **Text** permet d'éviter cette distinction.

Option Infer

Cette option indique si l'inférence des types des variables locales est active. Avec cette option, le compilateur détermine de lui-même le type des variables à partir des valeurs qui leur sont affectées.

Avertissements

Le compilateur est capable de détecter des problèmes potentiels dans votre code et de générer des avertissements associés. Vous avez la possibilité de configurer l'action entreprise par le compilateur pour différentes catégories de

problèmes. Les trois actions possibles sont :

- Aucun : le compilateur ignore le problème.
- Avertissement : le compilateur génère un avertissement dans la liste des tâches.
- Erreur : le compilateur génère une erreur de compilation.

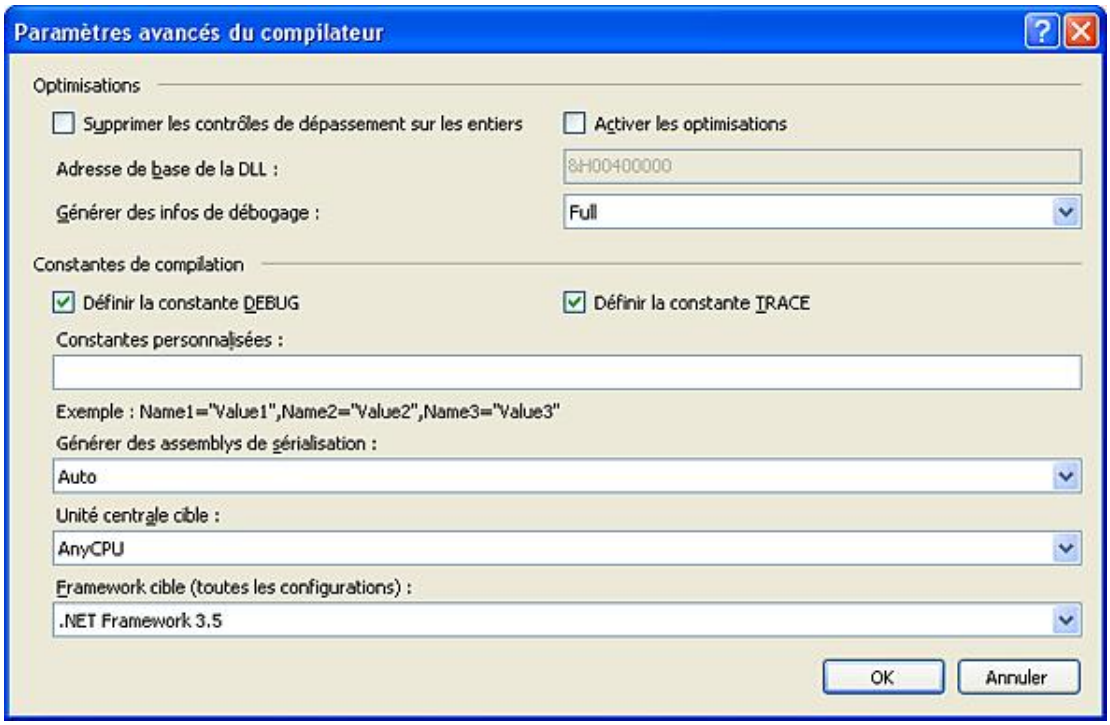
Vous pouvez également dévalider tous les warnings en utilisant l'option **Désactiver tous les avertissements** ou, au contraire, les traiter comme des erreurs avec l'option **Considérer tous les avertissements comme des erreurs**.

Générer le fichier de documentation XML

Avec cette option, le compilateur recherche dans le code les commentaires spéciaux placés grâce aux caractères ''' et les utilise pour générer le fichier de documentation. Ce fichier est créé dans le répertoire dans lequel est généré le fichier compilé.

Options avancées de compilation

Cette option propose une boîte de dialogue permettant la configuration avancée du compilateur.



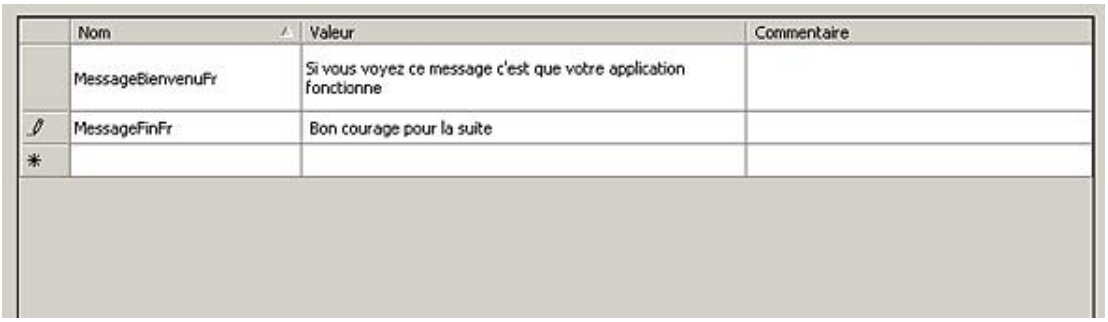
Cette boîte de dialogue permet la configuration des options suivantes :

- Supprimer la vérification de débordement lors de calculs sur des entiers.
- Autoriser les optimisations de code lors de la compilation.
- Spécifier l'adresse à laquelle une bibliothèque dll sera chargée.
- Indiquer si des informations de débogage sont ajoutées au résultat de la compilation.
- Définir les constantes de compilation.
- Demander la génération d'informations pour permettre la sérialisation XML.
- Indiquer un type de processeur spécifique pour l'exécution de cette application.

- Indiquer la version du framework utilisée pour l'exécution de l'application.

e. Ressources d'un projet

Les ressources sont utilisées pour externaliser certains éléments d'une application. Elles permettent de réaliser rapidement des modifications simples d'une application, sans avoir à rechercher dans des milliers de lignes de code. L'utilisation la plus classique consiste à séparer, du code, les constantes chaîne de caractères. Vous pouvez également créer des ressources icones, images, fichier texte, ou audio. Toutes les ressources sont gérées par cette boîte de dialogue.



- Pour chaque ressource, indiquez un nom et une valeur. Le nom sera bien sûr utilisé dans le code pour pouvoir récupérer la valeur.

En fonction du type de ressource, vous avez à votre disposition un éditeur adapté pour modifier la ressource. Les ressources peuvent être liées ou embarquées, en fonction de leur type. Une ressource liée est stockée dans son propre fichier et le fichier **Resources.resx** contient simplement un lien vers le fichier original. Une ressource embarquée est stockée directement dans le fichier **Resources.resx** de l'application. Dans tous les cas, les ressources seront compilées dans l'exécutable de l'application.

Voyons maintenant comment accéder aux ressources à partir du code de l'application. Toutes les ressources sont accessibles par la propriété **Resources de l'objet My**. L'exemple suivant utilise :

- Une ressource chaîne de caractères (MessageBienvenueFr)
- Une ressource icon (IconAppli)
- Une ressource image bitmap (ImageFond)
- Un fichier son (Musique)

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Load
Me.Icon = My.Resources.IconAppli
Me.BackgroundImage = My.Resources.ImageFond
My.Computer.Audio.Play(My.Resources.Musique, AudioPlayMode.BackgroundLoop)
MsgBox(My.Resources.MessageBienvenueFr)
End Sub
```

f. Paramètres d'application

Les paramètres d'application sont, en général, utilisés pour stocker et charger dynamiquement les paramètres de configuration d'une application, comme par exemple, les préférences de l'utilisateur ou les derniers fichiers utilisés dans l'application.

Les paramètres doivent d'abord être créés dans la page de propriétés suivante.

Pour chaque paramètre, vous devez fournir un nom, utilisé pour manipuler le paramètre dans le code et un type, pour le paramètre.

Vous devez également fournir une portée pour le paramètre. Deux choix sont possibles :

Utilisateur

Le paramètre peut être modifié pendant le fonctionnement de l'application.

Application

Le paramètre est en lecture seule pendant l'exécution et peut uniquement être modifié par l'intermédiaire de cette boîte de dialogue.

La dernière chose à faire est de spécifier une valeur pour le paramètre.

Nous allons maintenant étudier comment manipuler les paramètres dans le code. Nous devons réaliser trois opérations.

- Au démarrage de l'application, nous devons charger les paramètres. L'accès aux para- mètres se fait par la propriété Settings de l'objet My.

```
My.Settings.Reload()
```

- Pendant l'exécution de l'application, nous avons accès aux paramètres également par cette propriété Settings de l'objet My, à laquelle nous ajoutons le nom du paramètre. Ceci nous permet la lecture de la valeur du paramètre ou l'affectation d'une valeur au paramètre.

```
Me.BackColor = My.Settings.CouleurFond
My.Settings.DerniereUtilisation = Now
```

- À la fermeture de l'application, nous devons enfin sauvegarder les paramètres en utilisant la méthode Save :

```
My.Settings.Save()
```

Pour chaque utilisateur de l'application, une version distincte des paramètres est sauvegardée.

g. Autres paramètres de configuration

Les autres rubriques de configuration du projet concernant le déploiement de l'application sont traitées dans un chapitre spécifique.

Les variables, constantes et énumérations

1. Les variables

Les variables vont vous permettre de mémoriser, pendant l'exécution de votre application, différentes valeurs utiles pour le fonctionnement de votre application. Une variable doit obligatoirement être déclarée avant son utilisation dans le code. Lors de la déclaration d'une variable, vous fixez ses caractéristiques.

a. Nom des variables

Voyons les règles à respecter pour nommer les variables :

- le nom d'une variable commence obligatoirement par une lettre,
- il peut être constitué de lettres, de chiffres ou du caractère souligné (_),
- il peut contenir un maximum de 1023 caractères (pratiquement, il est préférable de se limiter à une taille plus raisonnable),
- il n'y a pas de distinction entre minuscules et majuscules (la variable AgeDuCapitaine et équivalente à ageducapitaine). Cependant, d'autres langages font une distinction entre minuscules et majuscules, aussi il faudra être prudent si vous générez un assembly utilisé par un autre langage.
- les mots clés du langage ne doivent pas être utilisés (c'est malgré tout possible mais dans ce cas, le nom de la variable doit être encadré par les caractères [et]. Par exemple, une variable nommée next sera utilisée dans le code sous cette forme [next]=56).

b. Type des variables

En spécifiant un type pour une variable, nous indiquons quelles informations nous allons pouvoir stocker dans cette variable.

Deux catégories de types de variables sont disponibles :

- Les types valeur : la variable contient réellement les informations.
- Les types référence : la variable contient l'adresse mémoire où se trouvent les informations.

Les différents types de variables disponibles sont définis au niveau du Framework lui-même. Vous pouvez également utiliser les alias définis au niveau de VB, peut-être plus explicites. Ainsi, le type **System.Int32** défini au niveau du framework peut être remplacé par le type **integer** dans Visual Basic.

Les différents types peuvent être classés en six catégories.

Les types numériques entiers

Types entiers signés			
Sbyte	- 128	127	8 bits
Short	-32768	32767	16 bits
Integer	-2 147 483 648	2 147 483 647	32 bits
Long	-9223372036854775808	9223372036854775807	64 bits
Types entiers non signés			

Byte	0	255	8 bits
UShort	0	65535	16 bits
UInteger	0	4294967295	32 bits
ULong	0	18446744073709551615	64 bits

Lorsque vous choisissez un type pour vos variables entières, vous devez prendre en compte les valeurs minimale et maximale que vous envisagez de stocker dans la variable afin d'optimiser la mémoire utilisée par la variable. Il est, en effet, inutile d'utiliser un type Long pour une variable dont la valeur n'excédera pas 50, un type Byte est dans ce cas suffisant.

L'économie de mémoire semble dérisoire pour une variable unique mais devient appréciable lors de l'utilisation de tableaux de grande dimension.

Si par contre vous souhaitez optimiser la vitesse d'exécution de votre code, il est préférable d'utiliser le type Integer.

Les types décimaux

Single	-3.40282347E+38	3.40282347E+38	4 octets
Double	-1.7976931348623157E+308	1.7976931348623157E+308	8 octets
Decimal	-79228162514264337593543950335	79228162514264337593543950335	16 octets

Les mêmes considérations d'optimisation que pour les variables entières doivent être prises en compte. Dans ce cas, une rapidité d'exécution maximale est obtenue avec le type Double. Le type Decimal est plus spécialement recommandé pour les calculs financiers pour lesquels les erreurs d'arrondis sont prohibées, mais au détriment de la rapidité d'exécution du code.

Les types caractères

Le type Char (caractères) est utilisé pour stocker un caractère unique. Une variable de type char utilise deux octets pour stocker le code Unicode du caractère. Dans jeu de caractère Unicode, les 128 premiers caractères sont identiques au jeu de caractère ASCII, les caractères suivants jusqu'à 255 correspondent aux caractères spéciaux de l'alphabet latin (par exemple, les caractères accentués), le reste est utilisé pour des symboles ou pour les caractères d'autres alphabets.

Pour pouvoir stocker des chaînes de caractères, il convient d'utiliser le type String, qui représente une suite de zéro à 2147483648 caractères. Les chaînes de caractères sont invariables car, lors de l'affectation d'une valeur à une chaîne de caractères, de l'espace est réservé en mémoire pour le stockage. Si, par la suite, cette variable reçoit une nouvelle valeur, le système lui assigne un nouvel emplacement en mémoire. Heureusement, ce mécanisme est transparent pour nous et la variable fera toujours automatiquement référence à la valeur qui lui a été assignée. Avec ce mécanisme, les chaînes de caractères peuvent avoir une taille variable. L'espace occupé en mémoire est automatiquement ajusté à la longueur de la chaîne de caractères.

Pour affecter une chaîne de caractères à une variable, le contenu de la chaîne doit être saisi entre " ", comme dans l'exemple ci-dessous :

Exemple

```
NomDuCapitaine = "Crochet"
```

De nombreuses fonctions permettent la manipulation des chaînes de caractères et seront détaillées plus loin dans ce chapitre. Il existe également un type String sous forme d'une classe permettant également la manipulation des chaînes de caractères. Dans ce cas, la manipulation se fera par les méthodes disponibles dans cette classe String. Elles seront également détaillées dans un paragraphe spécifique plus loin dans ce chapitre.

Le type Boolean

Le type Boolean permet d'utiliser une variable qui peut prendre deux états vrai/faux, oui/non, on/off.

L'affectation se fait directement avec les valeurs True ou False, comme dans l'exemple suivant :

```
Disponible=True
Modifiable=False
```

Il est toutefois possible d'affecter une valeur numérique à une variable de type Boolean. Dans ce cas, une valeur égale à zéro sera considérée comme un boolean False alors que toute autre valeur positive ou négative sera considérée comme un boolean True.

Le type Date

Le type Date permet de stocker, dans une variable, des informations concernant une date et une heure.

L'affectation se fait en encadrant la valeur par le signe # comme dans l'exemple suivant :

```
Aujourdhui=#10/27/2005 14 :58 :23#
```

Le format utilisé pour affecter une valeur à une variable de type date est toujours de la forme #mois/jour/année heure :minute :seconde# indépendamment du format de date configuré sur votre système d'exploitation. Les heures peuvent être spécifiées au format 12 heures ou 24 heures. De toute façon, l'environnement de développement surveille votre saisie et transformera toujours la date au format 12 heures. Ainsi, l'exemple précédent sera modifié par l'environnement de développement avec la forme suivante :

```
Aujourdhui=#10/27/2005 2:58:23 PM#
```

Si vous affectez, à une variable de type Date, une valeur contenant simplement une heure, Visual Basic considère qu'il s'agit du premier janvier de l'an 1 à l'heure que vous avez indiquée. Par exemple :

```
Jour=#12 :35 :30#
```

Correspond au premier janvier de l'an 1 à 12 heures 35 minutes 30 secondes.

Inversement si vous affectez, à une variable de type Date, une valeur contenant simplement une date, Visual Basic considère qu'il s'agit du jour que vous avez indiqué, à minuit.

Le type Object

C'est peut-être le type le plus universel de VB. Dans une variable de type Object, vous pouvez stocker n'importe quoi. En fait, ce type de variable ne stocke rien. La variable va contenir non pas la valeur elle-même, mais l'adresse, dans la mémoire de la machine, où l'on pourra trouver la valeur de la variable. Rassurez-vous, tout ce mécanisme est transparent et vous n'aurez jamais à manipuler les adresses mémoire directement.


Une variable de type Object pourra donc faire référence à n'importe quel autre type de valeur y compris des types numériques simples. Cependant, le code sera moins rapide du fait de l'utilisation d'une référence.

Les types Nullables

Il arrive parfois que dans certaines circonstances une variable n'a pas de valeur bien définie. C'est par exemple le cas lors de la récupération d'information en provenance d'une base de données si pour certains champs aucune valeur n'a été affectée dans la base. Comment représenter cette situation avec des variables dans Visual Basic. Une solution consiste à utiliser une valeur n'ayant aucune signification pour l'application. Par exemple, pour une variable numérique représentant un code postal dans l'application, il peut être envisagé d'affecter à cette variable une valeur négative dans le cas où le code postal n'est pas renseigné. Le reste du code doit bien sûr tenir compte de cette convention. Pour certains types d'informations cette solution n'est pas envisageable. Prenons le cas d'une variable de type Boolean pour lequel il n'y a que deux valeurs admises, 'true' ou 'false', comment représenter le fait que le contenu de la variable n'est pas 'non renseigné'.

Pour résoudre ce problème, Visual Basic propose les types Nullables. Ils permettent aux variables de type valeur de ne contenir aucune information. Pour activer cette fonctionnalité sur une variable il faut simplement utiliser le caractère '?' à la suite du nom de la variable ou de son type dans la déclaration comme dans l'exemple suivant.

```
Dim CodePostal ? as integer
```

ou

```
Dim CodePostal as integer?
```

Il faut par contre être prudent lors de l'utilisation d'une variable de ce type et vérifier avant de l'utiliser si elle contient effectivement une valeur. Pour cela, il faut tester la propriété `HasValue` de la variable pour déterminer si elle contient

effectivement une valeur. Si c'est le cas cette valeur est disponible via la propriété `Value` de la variable. Cette propriété est en lecture seule car l'affectation d'une valeur se fait directement sur la variable.

```
codePostal=17000
If codePostal.HasValue Then
   Console.WriteLine(codePostal)
Else
   Console.WriteLine("Code postal vide")
End If
```

Il est indispensable de tester la propriété `HasValue` avant l'utilisation de la propriété `value` car si la variable ne contient pas de valeur il y a déclenchement d'une exception. C'est le cas dans l'exemple ci-dessous puisqu'une variable nullable, contrairement à une variable normale, ne contient pas de valeur par défaut.

Une variable contenant une valeur peut revenir à l'état 'non renseigné' si on lui affecte la valeur `nothing`.

L'utilisation de variables de type boolean nullable avec les opérateurs logiques 'and' et 'or' peut parfois être problématique. Voici la table de vérité de ces deux opérateurs avec des variables nullables.

B1	B2	B1 and B2	B1 or B2
nothing	nothing	nothing	nothing
nothing	true	nothing	true
nothing	false	false	nothing
true	nothing	nothing	true
true	true	true	true
true	false	false	true
false	nothing	false	nothing
false	true	false	true
false	false	false	false

Le dernier point à éclaircir concerne l'utilisation d'un boolean nullable dans une structure conditionnelle. Regardons le code suivant :

```
Dim condition As Boolean?
…
If condition Then
   Console.WriteLine("le test est positif")
Else
   Console.WriteLine("le test est négatif")
End If
```

L'instruction 'if' considère que le test est positif uniquement si la variable contient la valeur 'true'. Pour les deux autres cas , valeur 'false' ou valeur 'non renseignée', le test est considéré comme négatif et le code du bloc 'else' s'exécute.

c. Conversions de types

Les conversions de types consistent à transformer une variable d'un type dans un autre type. Les conversions peuvent s'effectuer vers un type supérieur ou vers un type inférieur.

Si une conversion vers un type inférieur est utilisée, il risque d'y avoir une perte d'informations. Par exemple, la conversion d'un type Double vers un type Long fera perdre la partie décimale de la valeur.

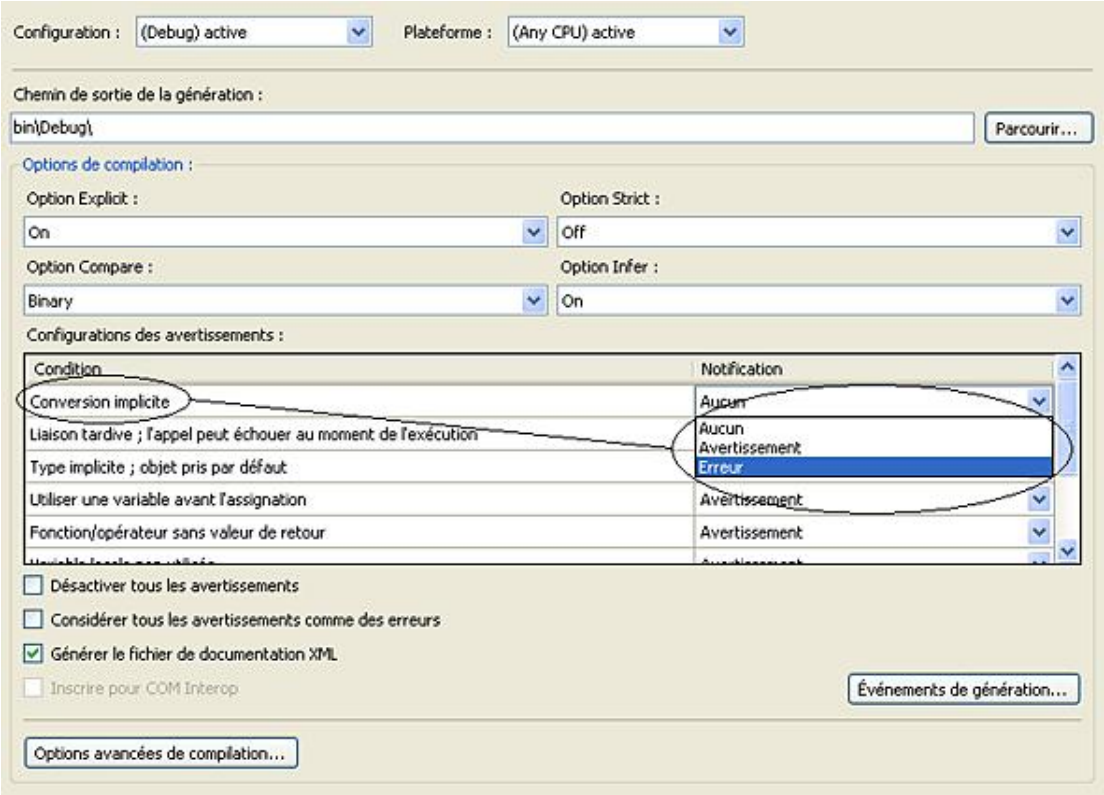
Exemple

```
Dim x As Double
Dim y As Long
x = 21.123456789012344
y = x
Console.WriteLine(" valeur de x : " & x)
Console.WriteLine(" valeur de y : " & y)
```

Affiche :

```
valeur de x : 21,1234567890123
valeur de y : 21
```

Une option du compilateur permet de vérifier l'existence de ce genre de conversion.



En fonction de la configuration de cette option, un warning ou une erreur de compilation sera générée si une telle conversion est tentée.

Elles peuvent également être implicites ou explicites en fonction du code utilisé. Les conversions implicites sont réalisées simplement par affectation d'une variable d'un type à une variable d'un autre type. Les conversions explicites nécessitent l'utilisation d'un mot clé spécifique qui sont de la forme Cxxx ou xxx correspondant au type dans lequel la valeur sera convertie.

Le tableau suivant résume ces différents opérateurs.

Opérateur	Type de destination
CBool	Boolean

CByte	Byte
CChar	Char
CDate	Date
CDbl	Double
CDec	Decimal
CInt	Integer
CLng	Long
CObj	Object
CSByte	SByte
CShort	Short
CSng	Single
CStr	String
CUint	UInteger
CULng	ULong
CUShort	UShort

Un autre opérateur (CType) permet la conversion vers un type standard du langage mais surtout vers un type personnalisé, comme par exemple, une classe.

Cet opérateur est plus général et il attend deux paramètres :

- la variable à convertir ;
- le type vers lequel doit se faire la conversion.

L'utilisation de ces opérateurs ne provoque pas d'erreur de compilation, si vous tentez une conversion restrictive, car le compilateur considère alors que vous la réalisez en toute connaissance de cause.

Les conversions à partir de chaînes de caractères et vers des chaînes de caractères sont plus spécifiques.

Conversion vers une chaîne de caractères

La fonction `format` permet de choisir la forme du résultat de la conversion d'une valeur quelconque en chaîne de caractères. Cette fonction attend comme paramètre la valeur à convertir et le format dans lequel vous souhaitez obtenir le résultat. Ce deuxième paramètre est fourni sous forme d'une chaîne de caractères exprimant l'aspect du résultat désiré.

Certains formats standards sont prédéfinis mais il est également possible de personnaliser le résultat de la fonction `format`. Ci-après, les paramètres de cette fonction sont présentés.

Formatage de valeurs numériques

Currency

Format monétaire tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format(12.35,"Currency")

```
Résultat : 12,35 €
```

Fixed

Utilise au moins un caractère pour la partie entière et au moins deux caractères pour la partie décimale d'un nombre. Le séparateur décimal est celui défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format(.2,"Fixed")

```
Résultat : 0,20
```

Percent

Multiplie la valeur indiquée par cent et ajoute le symbole " % " à la suite.

Exemple : format(.2,"Percent")

```
Résultat : 20,00%
```

Standard

Format numérique tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format(245813.5862,"Standard")

```
Résultat : 245 813,59
```

Scientific

Notation scientifique avec deux chiffres significatifs.

Exemple : format(245813.58,"Scientific")

```
Résultat : 2,46E+05
```

E

Notation scientifique avec six chiffres significatifs.

Exemple : format(245813.5862,"E")

```
Résultat : 2,458136E+005
```

X

Format hexadécimal. Utilisable uniquement pour les valeurs entières.

Exemple : format(245813,"X")

```
Résultat : 3C035
```

Yes/No

True/False

On/Off

Retourne No, False, Off si la valeur est égale à zéro sinon retourne Yes, True, On pour toutes les autres valeurs.

Chaîne de formatage personnalisée pour valeurs numériques

0

Réserve un emplacement pour un caractère numérique. Les zéros non significatifs sont affichés.

Exemple : format(245813.12,"00000000000.0000")

```
Résultat : 00000245813,1200
```

#

Réserve un emplacement pour un caractère numérique. Les zéros non significatifs ne sont pas affichés.

Exemple : format(245813.12,"###########.####")

```
Résultat : 245813,12
```

.

Réserve un emplacement pour le séparateur décimal. Le caractère réellement utilisé dans le résultat dépend de la configuration des options régionales et linguistiques du panneau de configuration du système.

,

Réserve un emplacement pour le séparateur de millier. Le caractère réellement utilisé dans le résultat dépend de la configuration des options régionales et linguistiques du panneau de configuration du système.

****

Permet l'utilisation d'un caractère ayant une signification spéciale comme caractère ordinaire dans une chaîne de formatage. Dans l'exemple suivant, le caractère \ fait perdre sa signification spéciale au caractère #

Exemple : format(325,"commande N\#0000")

```
Résultat : commande N#0325
```

Formats de date et heure

G

Format Date court et format Heure tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format (now,"G")

```
Résultat 17/10/2005 11:10:42
```

D

Format Date longue tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format (now,"D")

```
Résultat lundi 17 octobre 2005
```

d

Format Date court tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format (now,"d")

```
Résultat 17/10/2005
```

T

Format Heure tel que défini dans les options régionales et linguistiques du panneau de configuration du système.

Exemple : format (now,"T")

```
Résultat 11:45:30
```

s

Format 'triable '.

Exemple : format (now,"s")

```
Résultat 2005-10-17T11:47:30
```

Chaîne de formatage personnalisée pour valeurs de date et heure

d Jour du mois sans zéro non significatif

dd Jour du mois avec zéro non significatif

ddd Nom du jour de la semaine abrégé

dddd Nom du jour de la semaine complet

M Numéro du mois sans zéro non significatif

MM Numéro du mois avec zéro non significatif

MMM Nom du mois abrégé

MMMM Nom du mois complet

h Heure sans zéro non significatif (format 12H)

hh Heure avec zéro non significatif (format 12H)

H Heure sans zéro non significatif (format 24H)

HH Heure avec zéro non significatif (format 24H)

m Minute sans zéro non significatif

mm Minute avec zéro non significatif

s Seconde sans zéro non significatif

ss Seconde avec zéro non significatif

y Année sur un chiffre. Si, c'est le seul caractère de la chaîne de formatage, il faut dans ce cas utiliser %y

aayy Année sur deux chiffres

yyyy Année sur quatre chiffres

zzz Décalage par rapport au temps universel (GMT).

Conversion depuis une chaîne de caractères

La fonction `Val` permet la conversion d'une chaîne de caractères en valeur numérique. Elle lit la chaîne passée comme paramètre jusqu'à rencontrer un caractère autre qu'un chiffre, un espace, une tabulation, ou un point. Elle transforme ensuite cette portion de chaîne en valeur numérique, en tenant compte des éventuels paramètres de formatage définis au niveau du système, comme par exemple le séparateur de millier. Les caractères " &H " ou " &O " placés en

début de chaîne indiquent que la valeur est exprimée en hexadécimal ou en octal.

Exemple : val ("&H7FFF")

retourne

```
32767.0
```

d. Déclaration des variables

Par défaut, le compilateur Visual Basic considère que toute variable qui apparaît dans une application doit avoir été déclarée. Vous pouvez modifier cette option du compilateur en ajoutant dans votre code la ligne :

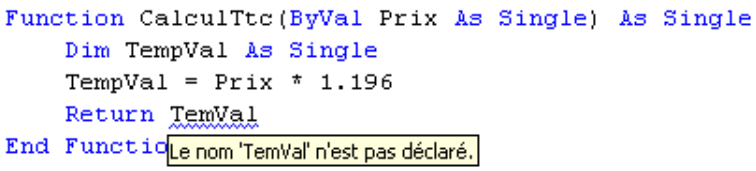
```
Option Explicit Off
```

Cette option doit être la première ligne du fichier source et s'appliquera alors à l'ensemble du code de ce fichier.

Avec cette option, vous n'avez plus l'obligation de déclarer une variable avant de l'utiliser. Bien que paraissant sympathique, cette solution risque de produire des erreurs difficiles à trouver dans votre code. Regardons le code suivant :

```
Function CalculTtc(Prix as Single) as Single
    Dim TempVal as Single
    TempVal=Prix * 1.196
Return TemVal
End Function
```

Pas de soucis au moment de la compilation et pourtant notre fonction nous retournera toujours une valeur égale à zéro. Si l'on regarde de plus près le code, on s'aperçoit d'une faute de frappe sur la ligne `Return Temval` : Il manque le p au nom de la variable `Tempval`. Lors de la compilation, Visual Basic a en fait considéré qu'il s'agissait d'une nouvelle variable et l'a donc initialisée à zéro. Noyé dans des centaines de lignes de code, ce genre d'erreurs peut être très difficile à détecter. En gardant l'option d'obligation des variables, vous auriez détecté le problème dès la compilation de votre application.



Voyons donc maintenant comment déclarer les variables en Visual Basic. L'instruction de base pour la déclaration d'une variable est l'instruction`Dim`.

La syntaxe générale de déclaration d'une variable est la suivante :

```
Dim NomVariable1[,NomVariable2,NonVariableN] [As Type de la Variable]
[= Valeur initiale]
```

Les paramètres entre crochets sont optionnels dans la déclaration. Si le type de la variable est omis, elle aura le type par défaut de vb c'est-à-dire **Object**.

Si la valeur initiale est omise, la variable sera initialisée à zéro si elle correspond à un type **numérique**, à une chaîne de caractère vide si elle est du type **String**, à la valeur Nothing si elle est du type **Object** et à false si elle est du type **Boolean**.

Si plusieurs noms sont spécifiés, les variables correspondantes seront toutes du type indiqué.

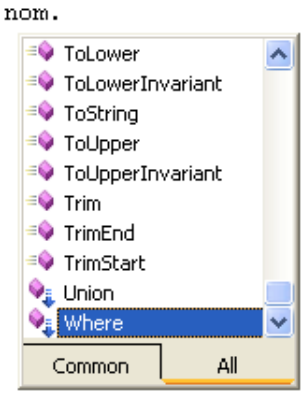
e. Inférence de type

Nous avons vu dans les sections précédentes qu'il est fortement souhaitable de toujours déclarer les variables avant leur utilisation. Cependant, dans certains cas, il est envisageable de laisser le compilateur réaliser une partie du travail. Grâce à l'inférence de type, le compilateur peut déterminer le type à utiliser pour une variable locale déclarée sans la clause 'as'. Pour cela, il se base sur le type de l'expression utilisée pour initialiser la variable. Dans l'exemple suivant la variable est considérée comme une chaîne de caractères.

```
Dim nom = "Dupont"
```

Pour s'assurer que cette variable est bien considérée comme une chaîne de caractères, il suffit de demander à

IntelliSense ce qu'il nous propose pour utiliser cette variable.

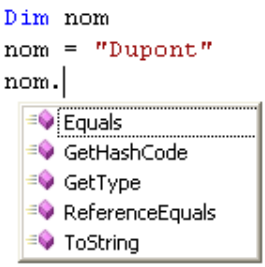


Nous avons bien à disposition les méthodes et propriétés du type String.

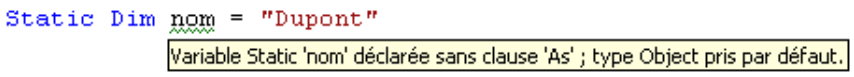
Pour que l'inférence de type fonctionne correctement, il faut impérativement respecter quelques règles :

- L'inférence ne fonctionne que pour les variables locales, c'est-à-dire les variables déclarées dans une procédure ou fonction.

- L'initialisation doit se faire sur la même ligne de code que la déclaration. Dans l'exemple ci-dessous la variable est considérée comme étant de type Object.



- La variable ne peut pas être déclarée static. Si c'est le cas, elle est considérée comme étant du type Object.



f. Portée des variables

La portée d'une variable est la portion de code à partir de laquelle l'on peut manipuler cette variable. Elle est fonction de l'emplacement où est située la déclaration et du mot clé utilisé pour la déclaration.

Portée niveau bloc

Seul le code du bloc aura la possibilité de travailler avec la variable (par exemple, dans une boucle `for next`). Toutefois, si le même bloc de code est exécuté plusieurs fois pendant l'exécution de la procédure ou fonction, cas d'une boucle `Do Loop` par exemple, la variable ne sera créée que lors du premier passage dans la boucle et conservera sa valeur d'un passage à l'autre dans la boucle. Pour modifier ce mode de fonctionnement, il suffit d'initialiser la variable au moment de sa déclaration. Seul le mot clé `Dim` peut être utilisé, pour la déclaration de la variable au sein d'un bloc.

Exemple :

```
Dim i As Integer
            For i = 0 To 5
                Dim j As Integer
                Dim k As Integer = 0
                j = j + 1
                k = k + 1
```

```
                Console.WriteLine("valeur de j :{0}", j)
                Console.WriteLine("valeur de k :{0}", k)
            Next
Affiche :
valeur de j :1
valeur de k :1
valeur de j :2
valeur de k :1
valeur de j :3
valeur de k :1
valeur de j :4
valeur de k :1
valeur de j :5
valeur de k :1
valeur de j :6
valeur de k :1
```

Portée niveau procédure

Seul le code de la procédure ou de la fonction où est déclarée la variable pourra modifier son contenu. Ce genre de variable est parfois nommé "variable locale". Seul le mot clé `Dim` peut être utilisé pour la déclaration d'une variable dans une procédure ou fonction.

Portée niveau module

Cette portée correspond, en fait, à une variable déclarée à l'extérieur d'une procédure ou fonction, c'est-à-dire dans un module, dans une classe ou dans une structure. La déclaration peut apparaître n'importe où dans le code (mais à l'extérieur de toute procédure ou fonction). Il est cependant recommandé de regrouper les déclarations afin de faciliter la maintenance ultérieure du code.

Portée niveau namespace

La variable sera utilisable à partir du code faisant partie du même namespace même si ce code est situé dans des modules différents. Si vous ne déclarez pas explicitement de namespace dans votre code, il y aura toujours le namespace par défaut, et on pourra dans ce cas parler plutôt de portée projet.

g. Niveau d'accès des variables

Le niveau d'accès d'une variable se combine avec la portée de la variable et détermine quelle portion de code a le droit de lire et d'écrire dans la variable. Un ensemble de mots clés permettent de contrôler le niveau d'accès. Ils s'utilisent à la place du mot clé `Dim` lors de la déclaration de la variable.

Public

Les éléments déclarés avec le mot clé `Public` seront accessibles de n'importe quelle portion de code du projet dans lequel ils sont déclarés et de n'importe quel autre projet référençant celui dans lequel ils sont déclarés. Le mot clé `Public` ne peut cependant pas être utilisé pour la déclaration à l'intérieur d'une procédure ou fonction.

Protected

Ce mot clé est utilisable uniquement à l'intérieur d'une classe. Il permet de restreindre l'accès à la variable, au code de la classe et au code de toutes les classes héritant de celle-ci.

Friend

Les éléments déclarés avec ce mot clé seront accessibles de l'assemblage dans lequel ils sont déclarés. Ce mot clé ne peut pas être utilisé à l'intérieur d'une procédure ou fonction.

Protected Friend

Ce niveau d'accès est l'union des niveaux d'accès Protected et Friend. Il rend visible la variable à l'ensemble de l'assemblage dans lequel elle est déclarée et à toutes les classes héritant de celle où elle est déclarée.

Private

Ce mot clé restreint l'accès à la variable au module, à la classe ou à la structure dans laquelle elle est déclarée. Il ne peut pas être utilisé à l'intérieur d'une procédure ou fonction.

Le mot clé `Dim` peut toutefois être utilisé partout pour la déclaration d'une variable, il faut se souvenir que dans le cas

d'une déclaration au niveau module, il est équivalent au mot clé `Private`. Encore une fois, pour faciliter la relecture du code, utilisez le mot clé `Private` au niveau module et réservez le mot clé `Dim` pour la déclaration de variables locales (à l'intérieur de procédures ou fonctions).

h. Durée de vie des variables

La durée de vie d'une variable nous permet de spécifier pendant combien de temps durant l'exécution de notre application le contenu de notre variable sera disponible.

Pour une variable déclarée dans une procédure ou fonction, la durée de vie correspond à la durée d'exécution de la procédure ou de la fonction. Dès la fin de l'exécution de la procédure ou fonction, la variable est éliminée de la mémoire. Elle sera recréée lors du prochain appel de la procédure ou fonction. Pour modifier ce mode de fonctionnement, il faut déclarer la variable avec le mot clé `static` à la place de `Dim`. Dans ce cas, lors du premier appel de la procédure ou fonction, la variable est créée en mémoire mais elle n'est pas détruite à la fin de la procédure ou fonction. Si la procédure ou fonction est à nouveau appelée dans le code, le même emplacement mémoire sera utilisé et la variable retrouvera son contenu précédent.

2. Les constantes

Dans une application, il arrive fréquemment que l'on utilise des valeurs numériques ou chaînes de caractères qui ne seront pas modifiées pendant le fonctionnement de l'application. Il est conseillé, pour faciliter la lecture du code, de définir ces valeurs sous forme de constantes.

La définition d'une constante s'effectue par le mot clé `const`.

Exemple

```
Const ValeurMaxi = 100
Const Message="Trop grand"
```

La constante peut être alors utilisée dans le code à la place de la valeur qu'elle représente.

```
If resultat> ValeurMaxi then
Console.WriteLine(Message)
```

Il sera parfois nécessaire de spécifier un type pour la constante que l'on déclare (dans le cas où le compilateur est configuré en mode Strict). La déclaration de la constante aura alors la forme suivante :

```
Const  ValeurMaxi as integer =100
```

Les règles concernant la durée de vie et la portée des constantes sont identiques à celles concernant les variables.

La valeur d'une constante peut également être calculée à partir d'une autre constante.

Exemple

```
Public Const Total As Integer = 100
Public Const Demi As Integer = Total / 2
```

Dans ce cas de figure, il faut être prudent et ne pas créer de référence circulaire qui provoquerait une erreur de compilation.

```
Public Const total As Integer = demi * 2
Public Const demi As Integer = total / 2
```

La constante 'total' ne peut pas dépendre de sa propre valeur.

De nombreuses constantes sont déjà définies au niveau du langage Visual Basic. Le nom de ces constantes commence en général par vb...

Les plus utilisées sont reprises dans le tableau suivant :

Nom de la constante	Valeur de la constante
vbCr	Retour Chariot (caractère N°13)
vbLf	Saut de ligne (caractère N°10)
vbCrLf	Combinaison Retour chariot + saut de ligne
vbNullChar	Caractère Null (caractère N°0)
vbTab	Tabulation (caractère N°9)
vbBack	Retour arrière (caractère N°8)

3. Les énumérations

Une énumération va nous permettre de définir un ensemble de constantes qui sont liées entre elles. La déclaration s'effectue de la manière suivante :

```
Enum jours
  Dimanche
  Lundi
  Mardi
  Mercredi
  Jeudi
  Vendredi
  Samedi
End Enum
```

Par défaut, la première valeur de l'énumération est initialisée à zéro. Les constantes suivantes sont ensuite initialisées avec un incrément de un. La déclaration précédente aurait donc pu s'écrire :

```
Const Dimanche = 0
Const Lundi = 1
Const Mardi = 2
Const Mercredi = 3
Const Jeudi = 4
Const Vendredi = 5
Const Samedi = 6
```

La séquence d'incrémentation automatique dans une énumération peut être interrompue, voire ne pas être utilisée comme dans l'exemple suivant :

```
Enum dalton
  Joe = 158
  Jack = 163
  William = 173
  Averell = 185
End Enum
```

Il faut toutefois que les valeurs utilisées dans l'énumération soient des valeurs entières.

Une fois définie, une énumération peut ensuite être utilisée comme un type de variable spécifique.

```
Dim Taille as Dalton
```

Les seules valeurs que vous pouvez affecter à votre variable **Taille** sont celles qui sont définies dans l'énumération.

```
Taille Dalton.Joe
Console.WriteLine(Taille)
Taille =45 ' Invalide
```

Lorsque vous faites référence à un élément de votre énumération, vous devez le faire précéder du nom de l'énumération comme dans l'exemple précédent. Pour éviter cela, il faut spécifier dans votre code que vous voulez utiliser l'énumération en l'important à l'aide de l'instruction suivante :

```
Imports Application.dalton
```

Il est alors possible d'utiliser les valeurs contenues dans l'énumération directement.

```
Taille = Joe
```

Il faut, dans ce cas, être prudent avec des noms de constantes identiques qui pourraient exister dans des énumérations différentes.

La déclaration d'une énumération ne peut pas se faire dans une procédure ou une fonction.

La portée d'une énumération suit les mêmes règles que celle des variables (utilisation des mots clés Public, Private, Friend, Protected).

4. Les tableaux

Les tableaux vont nous permettre de faire référence à un ensemble de variables par le même nom et d'utiliser un index pour les différencier. Un tableau peut avoir une ou plusieurs dimensions (jusqu'à 32, mais au-delà de trois on a du mal à se représenter le contenu du tableau). Le premier élément d'un tableau a toujours pour index, zéro.

L'index maximum d'un tableau est spécifié au moment de la création du tableau. Le nombre d'éléments d'un tableau est donc égal au plus grand index plus un.

Tableaux à une dimension

La déclaration s'effectue comme une variable classique mis à part que l'on indique à la suite du nom de la variable entre parenthèses le plus grand index du tableau :

```
Dim ChiffreAffaire(11) as Decimal
```

Cette déclaration va créer un tableau avec douze cases numérotées de 0 à 11. Les éléments des tableaux sont accessibles de la même manière qu'une variable classique. Il suffit d'ajouter l'index de l'élément que l'on veut modifier.

```
ChiffreAffaire(1)=12097
```

Une autre solution est disponible pour la création d'un tableau. Elle permet simultanément la création du tableau et l'initialisation de son contenu. La syntaxe est la suivante :

```
Dim tauxTva() As Decimal = {0, 5.5, 19.6, 33}
```

Il n'est dans ce cas nul besoin de préciser de taille pour le tableau. Le dimensionnement se fera automatiquement en fonction du nombre de valeurs placées entre les accolades.

Tableaux à plusieurs dimensions

La syntaxe de déclaration est similaire à celle d'un tableau, mis à part que l'on doit spécifier une valeur pour le plus grand index de chacune des dimensions du tableau en les séparant par une virgule.

```
Dim Cube(4, 4, 4) As Integer
```

L'accès à un élément du tableau s'effectue de manière identique, en indiquant les index permettant d'identifier la casse du tableau concernée.

```
Cube(0,1,1)=52
```

La syntaxe permettant l'initialisation d'un tableau à plusieurs dimensions au moment de sa déclaration est un petit peu plus complexe.

```
Dim Grille(,) As Integer = {{1, 2}, {3, 4}}
```

Cet exemple crée un tableau à deux dimensions de deux cases sur deux cases.

La création, avec cette technique, de tableaux de grande taille à plusieurs dimensions risque d'être périlleuse.

Redimensionnement d'un tableau

La taille des tableaux n'est pas figée mais peut être modifiée pendant le fonctionnement de l'application. L'instruction `Redim` permet de modifier la taille d'un tableau.

```
Redim ChiffreAffaire(52)
```

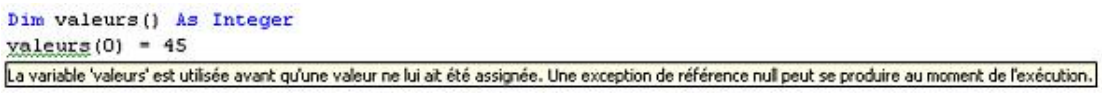
Nous avons maintenant cinquante trois cases disponibles pour stocker l'information dans notre tableau. Par contre, si notre tableau contenait des informations, elles ont été perdues pendant le redimensionnement. Pour conserver son contenu, il convient de spécifier le mot clé `Preserve` après l'instruction `Redim`.

```
Redim Preserve ChiffreAffaire(52)
```

Dans ce cas, le contenu initial du tableau est conservé et de nouvelles cases sont ajoutées à la suite de celles existant déjà, sans en affecter le contenu.

Dans le cas d'un tableau multidimensionnel, seule la dernière dimension peut être modifiée si vous souhaitez conserver le contenu du tableau. Le nombre de dimensions d'un tableau ne peut, par contre, pas être modifié par une instruction `Redim`.

L'instruction `Redim` peut également être utilisée pour fournir une taille initiale à un tableau. Dans ce cas, il ne faut pas spécifier de taille pour le tableau au moment de sa déclaration et utiliser l'instruction `Redim` pour le dimensionner. Il est, dans ce cas, impératif d'utiliser l'instruction `Redim` avant tout accès au contenu du tableau.



Manipulations courantes des tableaux

Lorsque l'on travaille avec les tableaux, certaines opérations doivent être fréquemment réalisées. Ce paragraphe décrit les opérations les plus courantes réalisées sur les tableaux.

Obtenir la taille d'un tableau

Il suffit d'utiliser la propriété `Length` du tableau pour connaître le nombre d'éléments qu'il peut contenir. Dans le cas d'un tableau multidimensionnel, le résultat correspond au nombre total de cases du tableau soit le produit de la taille de chacune des dimensions. Il ne s'agit pas de l'espace mémoire occupé par le tableau mais bien du nombre total de cases du tableau.

```
Dim Grille(,) As Integer = {{1, 2}, {3, 4}}
Console.WriteLine("taille totale du tableau : {0}", Grille.Length)
```

Pour obtenir l'occupation mémoire du tableau, il faut multiplier sa taille par le nombre d'octets utilisés pour une case élémentaire du tableau.

Obtenir la taille d'une des dimensions d'un tableau

La méthode `GetLength` attend comme paramètre la dimension du tableau pour laquelle l'on souhaite obtenir la taille :

```
Dim Matrice(,) As Integer = {{1, 2}, {3, 4}, {5, 6}}
Console.WriteLine("taille de la premiere dimension : {0}", Matrice.GetLength(0))
Console.WriteLine("taille de la deuxieme dimension : {0}", Matrice.GetLength(1))
```

Affiche le résultat suivant :

```
taille de la premiere dimension : 3
taille de la deuxieme dimension : 2
```

Obtenir la dimension d'un tableau

La propriété `Rank` d'un tableau renvoie directement la dimension du tableau :

```
Dim Grille(,) As Integer = {{1, 2}, {3, 4}, {5, 6}}
Console.WriteLine(" ce tableau comporte {0} dimensions",
Grille.Rank)
```

Affiche le résultat suivant :

```
ce tableau comporte 2 dimensions
```

Rechercher un élément dans un tableau

La fonction `IndexOf` de la classe `Array` permet d'effectuer une recherche dans un tableau. Elle accepte comme paramètres, le tableau dans lequel se fait la recherche et l'élément recherché dans le tableau. La valeur retournée correspond à l'index où l'élément a été trouvé dans le tableau ou -1 si élément ne se trouve pas dans le tableau.

```
Dim gouter() As String = {"pain", "beurre", "moutarde", "confiture"}
Console.WriteLine(Array.IndexOf(gouter, "moutarde"))
```

Trier un tableau

La procédure `Sort` de la classe `Array` assure le tri du tableau qu'elle reçoit en paramètre. Le tri s'effectue par ordre alphabétique pour les tableaux de chaîne de caractères et par ordre croissant pour les tableaux de valeurs numériques.

```
Dim gouter() As String = {"pain", "beurre", "moutarde", "confiture"}
Dim plat As String
Array.Sort(gouter)
For Each plat In gouter
       Console.WriteLine(plat)
Next
```

Affiche le résultat suivant :

```
beurre
confiture
moutarde
pain
```

5. Les chaînes de caractères

Les variables de type String permettent la manipulation de chaînes de caractères par votre application. Nous avons le choix entre deux possibilités pour travailler avec les chaînes de caractères :

- Utiliser les fonctions de Visual Basic
- Utiliser les méthodes de la classe System.String

Nous allons regarder comment réaliser les opérations les plus courantes sur les chaînes de caractères.

Affectation d'une valeur à une chaîne

Nous avons vu que pour, affecter une valeur à une chaîne, il faut la spécifier entre les caractères "et", un problème se pose si nous voulons que le caractère " fasse partie de la chaîne. Pour qu'il ne soit pas interprété comme caractère de début ou de fin de chaîne, il faut doubler le caractère " comme dans l'exemple ci-dessous.

```
Dim Chaîne as String
Chaîne=" il a dit : "" ça suffit ! """
Console.WriteLine(Chaîne)
```

Nous obtenons à l'affichage : il a dit : "ça suffit ! "

Pour les exemples suivants, nous allons travailler avec deux chaînes.

```
chaîne1 = "l'hiver sera pluvieux"
```

```
chaîne2 = "l'hiver sera froid"
```

Extraction d'un caractère particulier

Pour obtenir le caractère présent à une position donnée d'une chaîne de caractères, l'on peut considérer la chaîne comme un tableau de caractères et ainsi atteindre le caractère souhaité par un index.

```
Console.WriteLine("Le troisième caractère de la chaîne1 est : {0}", chaîne1(2))
```

Autre solution, mais cette fois en utilisant la propriété `Chars` de la classe `String`.

```
Console.WriteLine("Le troisième caractere de la chaîne1 est : {0}",
chaîne1.Chars(2))
```

Résultat :

```
Le troisième caractère de la chaîne est : h
```

Dans les deux cas, la numérotation des caractères commence à zéro comme pour un tableau.

Obtention de la longueur d'une chaîne

Pour déterminer la longueur d'une chaîne, la propriété `Length` de la classe `String` ou la fonction `Len` de Visual Basic sont disponibles. Toutes deux retournent le nombre de caractères présents dans la chaîne.

```
Console.WriteLine("la chaîne1 contient {0} caractères", chaîne1.Length)
Console.WriteLine("la chaîne2 contient {0} caractères", Len(chaîne2))
```

Résultat :

```
La chaîne 1 contient 21 caractères.
La chaîne 2 contient 18 caractères.
```

Découpage de chaîne

Plusieurs solutions sont disponibles en fonction de la portion de chaîne que l'on souhaite récupérer.

Le début de la chaîne

La fonction `Left` renvoie les x premiers caractères de la chaîne.

```
Console.WriteLine("les cinq premiers caractères de la chaîne1 sont {0}",
Left(chaîne1,5))
```

Résultat :

```
Les cinq premiers caractères de la chaîne sont l'hiv.
```

La fin de la chaîne

La fonction `Right` renvoie les x derniers caractères de la chaîne.

```
Console.WriteLine("les cinq derniers caractères de la chaîne1 sont {0}",
Right(chaîne1, 5))
```

Résultat :

```
les cinq derniers caractères de la chaîne sont froid.
```

Une portion quelconque de la chaîne

La fonction `Mid` ou la méthode `Substring` de la classe `String` retournent une portion de chaîne en fonction de la position de départ et du nombre de caractères à retourner qui leur sont passés. La fonction `Mid` nécessite en plus la chaîne à partir de laquelle va s'effectuer le découpage.

```
Console.WriteLine("Un morceau de la chaîne1 {0}", Mid(chaîne1, 2, 5))
Console.WriteLine("Un morceau de la chaîne2 {0}", chaîne2.Substring(2, 5))
```

Nous obtenons à l'affichage :

```
Un morceau de la chaîne1 'hive
Un morceau de la chaîne2 hiver
```

Attention pour la fonction Mid, la numérotation des caractères commence à 1.

Comparaison de chaînes

Plusieurs solutions sont possibles selon l'objectif à atteindre avec la comparaison des deux chaînes. Si le but est seulement de vérifier l'égalité de deux chaînes, vous pouvez utiliser l'opérateur = ou la fonction `Equals` de la classe `String`.

```
        If chaîne1 = chaîne2 Then
            Console.WriteLine("ce sont  les mêmes")
        Else
            Console.WriteLine("ce ne sont pas les mêmes")
        End If

       If chaîne1.Equals(chaîne2) Then
            Console.WriteLine("ce sont les mêmes")
        Else
            Console.WriteLine("ce ne sont pas les mêmes")
        End If
```

Pour réaliser un classement, vous devez par contre utiliser la méthode `Compare` de la classe `String` ou la fonction `StrComp`. Avec ces deux solutions, les deux chaînes à comparer doivent être passées comme paramètres. Le résultat de la comparaison est retourné sous forme d'un entier inférieur à zéro si la première chaîne est inférieure à la deuxième, égal à zéro si les deux chaînes sont identiques, et supérieur à zéro si la première chaîne est supérieure à la deuxième.

```
Select Case chaîne1.CompareTo(chaîne2)
                Case Is < 0
                    Console.WriteLine("chaîne1 est inférieure à chaîne2")
                Case 0
                    Console.WriteLine("chaîne1 est égale à chaîne2")
                Case Is > 0
                    Console.WriteLine("chaîne1 est supérieure à chaîne2")
 End Select
```

Insertion dans une chaîne

La méthode `insert` de la classe `String` permet l'insertion d'une chaîne dans une autre. Elle attend comme paramètre un entier et une chaîne et nous retourne la chaîne de départ dans laquelle se trouve insérée, à la position spécifiée, la chaîne passée en paramètre.

```
Dim chaîne3 As String
            chaîne3 = chaîne2.Insert(13, "tres ")
            Console.WriteLine(chaîne3)
```

L'instruction précédente nous affiche la ligne correspondante :

```
l'hiver sera tres froid
```

Suppression des espaces

- au début de la chaîne : `Console.WriteLine(ltrim(chaîne1))` Ou `Console.WriteLine(chaîne1.TrimStart())`
- À la fin de la chaîne : `Console.WriteLine(rtrim(chaîne1))` Ou `Console.WriteLine(chaîne1.TrimEnd())`
- Au début et à la fin : `Console.WriteLine(trim(chaîne1))` Ou `Console.WriteLine(chaîne1.Trim())`

Changement de la casse

- Tout en majuscules : `Console.WriteLine(Ucase(chaîne1))` Ou `Console.WriteLine(chaîne1.ToUpper())`
- Tout en minuscules : `Console.WriteLine(Lcase(chaîne1))` Ou `Console.WriteLine(chaîne1.ToLower())`

Recherche dans une chaîne

La fonction `InStr` ou la méthode `IndexOf` de la classe `String` permettent la recherche d'une chaîne à l'intérieur d'une autre. Le premier paramètre correspond à la chaîne dans laquelle va s'effectuer la recherche, le deuxième paramètre correspond à la chaîne recherchée. La fonction retourne un entier indiquant la position à laquelle la chaîne a été trouvée ou zéro si la chaîne n'a pas été trouvée. Par défaut, la recherche commence au début de la chaîne, sauf si vous utilisez une autre version de la fonction `InStr` qui, elle, attend trois paramètres, le premier paramètre étant pour cette fonction, la position de départ de la recherche, les deux autres étant respectivement la chaîne dans laquelle va s'effectuer la recherche et la chaîne recherchée. La numérotation des caractères se fait à partir de 1 pour la fonction `InStr`.

```
Dim recherche As String
            Dim position As Integer
            recherche = "e"
            position = InStr(chaîne1, recherche)
            While (position > 0)
                Console.WriteLine("chaîne trouvée à la position {0}", position)
                position = InStr(position + 1, chaîne1, recherche)
            End While
            Console.WriteLine("fin de la recherche")
```

Nous obtenons à l'affichage :

```
chaîne trouvée à la position 6
chaîne trouvée à la position 10
chaîne trouvée à la position 19
fin de la recherche
```

Remplacement dans une chaîne

Il est parfois souhaitable de pouvoir rechercher la présence d'une chaîne à l'intérieur d'une autre, comme dans l'exemple précédent, mais également de remplacer les portions de chaînes trouvées. La fonction `Replace` permet de spécifier une chaîne de substitution pour la chaîne recherchée. Elle attend au total cinq paramètres :

- La chaîne dans laquelle va s'effectuer la recherche.
- La chaîne recherchée
- La chaîne de remplacement
- La position de départ de la recherche
- Le nombre de remplacements souhaités (-1 pour toutes les occurrences de la chaîne recherchée)

```
chaîne3 = Replace(chaîne1, "hiver", "ete", 1, -1)
Console.WriteLine(chaîne3)
```

Nous obtenons à l'affichage :

```
l'ete sera pluvieux
```

6. Les structures

Les structures offrent la possibilité de combiner des données de différents types pour créer un nouveau type

composite. Ce nouveau type pourra ensuite être utilisé dans la déclaration de variables comme un type standard de Visual Basic.

Les structures sont très pratiques lorsque l'on souhaite manipuler des informations ayant un lien entre elles. Par exemple, dans une application comptable, les informations concernant les clients (code client, nom, prénom, adresse) peuvent être plus facilement gérées sous forme d'une structure plutôt que par des variables individuelles.

a. Déclaration d'une structure

La déclaration d'une structure se fait entre les mots clés `Structure` et `End Structure`. Entre ces deux mots clés, vous devez placer au moins la déclaration d'un membre de la structure. Les membres de la structure sont, en fait, tout simplement des variables, procédures ou fonctions déclarés à l'intérieur de la structure. Comme pour tout élément déclaré dans Visual Basic vous avez la possibilité de spécifier un niveau d'accès pour chaque membre de la structure. Sans information spécifique, le membre est considéré comme étant public. Par contre, il est impossible d'initialiser les membres d'une structure au moment de la déclaration.

Exemple

```
Public Structure Client
            Public Code As Integer
            Public Nom As String
            Public Prenom As String
            Public Coordonnees As String
End Structure
```

Les membres d'une structure peuvent être eux-mêmes des variables de type structure. Dans l'exemple précédent la variable coordonnées peut être décomposée sous forme d'une structure de type adresse.

Exemple

```
        Public Structure Adresse
            Public Numero As Integer
            Public Rue As String
            Public CodePostal As Integer
            Public Ville As String
        End Structure
        Public Structure Client
            Public Code As Integer
            Public Nom As String
            Public Prenom As String
            Public Coordonnees As Adresse
        End Structure
```

Les structures acceptent également des procécures ou fonctions comme membres. Elles sont généralement utilisées pour manipuler les variables membres de la structure.

Exemple

```
Public Structure Adresse
Public Numero As Integer
Public Rue As String
Public CodePostal As Integer
Public Ville As String
Public Function getAdresse () As String
      Return Numero & " " & Rue & vbCrLf & CodePostal & vbTab & Ville.ToUpper
End Function
End Structure
```

b. Utilisation des structures

Les structures sont utilisées comme des types de données classiques. Il convient, au préalable, de déclarer une variable du type de la structure.

```
Dim Client1 As Client
```

Par la suite, cette variable permet l'accès aux membres de la structure grâce à l'opérateur "." appelé opérateur d'accès.

Exemple

```
Client1.Code = 999
Client1.Nom = "LeNom"
Client1.Prenom = "lePrenom"
```

Si l'un des membres de la structure est lui-même de type structure, vous devez également utiliser l'opérateur d'accès pour pouvoir manipuler les membres imbriqués.

Exemple

```
Client1.Coordonnes.Numero = 42
Client1.Coordonnes.Rue = "rue de Paris"
Client1.Coordonnes.CodePostal = 44000
Client1.Coordonnes.Ville = "Nantes"
```

Les types structure sont des types de données par valeur, c'est-à-dire que la variable contient vraiment les données de la structure (à mettre en opposition avec les variables de type référence où la variable ne contient que l'emplacement, dans la mémoire, où se trouvent les données).

Ceci nous permet d'assigner à une variable de type structure le contenu d'une autre variable de même type. Il y a, dans ce cas, recopie des informations de chaque membre de la variable source dans le membre correspondant de la variable destination.

Exemple

```
Dim Client2 As Client
            Client2 = Client1
            Console.WriteLine("Client1 : " & vbCrLf & Client1.etiquette)
            Console.WriteLine("Client2 : " & vbCrLf & Client2.etiquette)
```

Affiche le résultat suivant :

```
Client1 :
M lePrenom LeNom
42 rue de Paris
44000 NANTES

Client2 :
M lePrenom LeNom
42 rue de Paris
44000 NANTES
```

Il faut cependant être méfiant avec ce mécanisme si un membre de la structure est un type référence (un tableau par exemple), car dans ce cas les deux variables se partageront le même tableau.

Les opérateurs

Les opérateurs sont des mots clés du langage permettant l'exécution d'opérations sur le contenu de certains éléments, en général des variables, des constantes, des valeurs littérales, ou des retours de fonctions. La combinaison d'un ou plusieurs opérateurs et d'éléments sur lesquels les opérateurs vont s'appuyer se nomme une expression. Ces expressions sont évaluées au moment de leur exécution, en fonction des opérateurs et des valeurs qui sont associées.

Les opérateurs peuvent être répartis en six catégories.

1. Les opérateurs d'affectation

Le seul opérateur disponible dans cette catégorie est l'opérateur =. Il permet d'affecter une valeur à une variable. Le même opérateur est utilisé quel que soit le type de la variable (numérique, chaîne de caractères...).

2. Les opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des calculs sur le contenu des variables :

Opérateur	Opération réalisée	Exemple	Résultat
++	Addition	6+4	10
-	Soustraction	12-6	6
*	Multiplication	3*4	12
/	Division	25/3	8.3333333333
\	Division entière	25/3	8
Mod	Modulo (reste de la division entière)	25 mod 3	1
^^	Puissance	5 ^ 3	125

3. Les opérateurs binaires

Ces opérateurs effectuent des opérations sur des entiers uniquement (Byte, Short, Integer, Long). Ils travaillent au niveau du bit sur les variables qu'ils manipulent.

Opérateur	Opération réalisée	Exemple	Résultat
And	Et Binaire	45 and 255	45
Or	Ou Binaire	99 or 46	111
Xor	Ou exclusif	99 xor 46	77
Not	Négation	Not 23	-24

4. Les opérateurs de comparaison

Les opérateurs de comparaison sont utilisés dans les structures de contrôle d'une application (if then, do loop...). Ils renvoient une valeur de type boolean en fonction du résultat de la comparaison effectuée. Cette valeur sera ensuite

utilisée par la structure de contrôle.

Opérateur	Opération réalisée	Exemple	Résultat
=	Égalité	2 = 5	False
	Inégalité	2 <> 5	True
<	Inférieur	2 < 5	True
>	Supérieur	2 > 5	False
<=	Inférieur ou égal	2 <= 5	True
>=	Supérieur ou égal	2 >= 5	False
Like	Égalité de chaînes de caractères	“toto” like “t*”	True
Is	Comparaison de deux variables objets	O1 Is O2	True si O1 et O2 font référence au même objet
IsNot	Comparaison de deux variables objets	O1 IsNot O2	True si O1 ne fait pas référence au même objet que O2
TypeOf ... Is ...	Comparaison du type de la variable avec le type donné	TypeOf O1 Is Client	True si la variable O1 référence un objet créé à partir du type Client

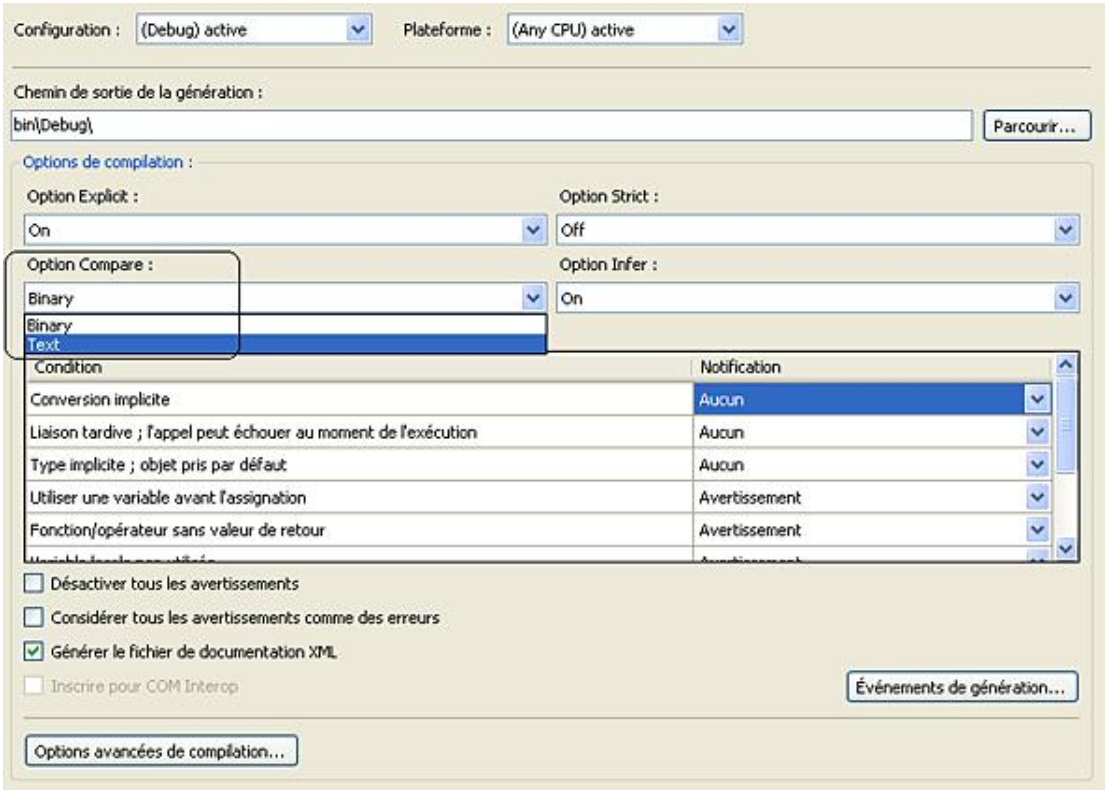
Quelques informations supplémentaires concernant l’opérateur like :

- Comme l’opérateur =, l’opérateur like permet de tester l’égalité de deux chaînes de caractères mais il est plus souple d’utilisation grâce à l’utilisation de caractères “jokers”.

- Le caractère * permet de remplacer n’importe quelle suite de caractères. L’expression `nom like "t*"` nous permet de vérifier si la variable `nom` contient une valeur commençant par t et ayant un nombre quelconque de caractères.

- Le caractère ? permet de remplacer un caractère dans l’expression à tester. Pour vérifier si votre variable `nom` contient une valeur commençant par t et ayant quatre caractères on utilisera l’expression suivante : `Nom like "t???"`

Il faut également être prudent avec les comparaisons de chaînes de caractères car les résultats d’une même comparaison peuvent être différents en fonction de l’option choisie pour le compilateur. Les options du compilateur peuvent être modifiés par la boîte de dialogue des propriétés de votre projet.



La propriété `Option Compare` permet de spécifier le mode de comparaison utilisé par le compilateur.

- Avec l'option **Text**, le compilateur ne fera pas de différence entre les minuscules et les majuscules lors de la comparaison.
- L'option **binaire** exigera une stricte égalité des chaînes pour obtenir un résultat true.

5. Les opérateurs de concaténation

Deux opérateurs + et & peuvent être utilisés pour la concaténation de chaînes de caractères. Cependant, la destination première de l'opérateur + est l'addition de valeurs numériques. Il possède un fonctionnement assez complexe en cas d'utilisation sur des chaînes de caractères. Les règles suivantes sont appliquées :

- Si les deux opérandes sont numériques, alors il y a addition.
- Si les deux opérandes sont de type String alors il y a concaténation.
- Si un opérande est numérique et le deuxième est de type String alors un autre paramètre entre en jeu. Si le compilateur est configuré avec l'option `Strict On` une erreur de compilation est générée. Si le compilateur est configuré avec l'option `Strict Off`, la chaîne de caractères est implicitement convertie en type Double puis additionnée. Si la chaîne de caractères ne peut pas être convertie en Double alors une exception est générée.

Pour éviter de se poser toutes ces questions, préférez l'utilisation de l'opérateur &. Avec cet opérateur, si l'un des deux opérandes n'est pas du type String, il est automatiquement converti vers ce type avant la concaténation.

L'inconvénient de l'opérateur & est qu'il n'est pas très rapide. Si vous avez de nombreuses concaténations à exécuter sur une chaîne, il est préférable d'utiliser la classe StringBuilder.

Exemple

```
Dim duree, i As Integer
            Dim lievre As String
            Dim tortue As String
            Dim debut, fin As Date
            debut = Now
```

```
            For i = 0 To 100000
                tortue = tortue & " " & i
            Next
            fin = Now
            duree = DateDiff(DateInterval.Second, debut, fin)
            Console.WriteLine("durée pour la tortue : {0} secondes", duree)
            debut = Now
            Dim sb As New StringBuilder
            For i = 0 To 100000
                sb.Append(" ")
                sb.Append(i)
            Next
            lievre = sb.ToString
            fin = Now
            duree = DateDiff(DateInterval.Second, debut, fin)
            Console.WriteLine("durée pour le lièvre : {0} secondes", duree)
            If lievre.Equals(tortue) Then
                Console.WriteLine("les deux chaînes sont identiques")
            End If
```

Résultat de la course :

```
durée pour la tortue : 107 secondes
durée pour le lièvre : 0 secondes
les deux chaînes sont identiques
```

Ce résultat se passe de commentaire !

6. Les opérateurs logiques

Les opérateurs logiques permettent de combiner les expressions dans des structures conditionnelles ou de boucle.

Opérateur	**Opération**	**Exemple**	**Résultat**
and	Et logique	If (test1) And (test2)	vrai si test1 **et** test2 est vrai
Or	Ou logique	If (test1) Or (test2)	vrai si test1 **ou** test2 est vrai
xor	Ou exclusif	If (test1) Xor (test2)	vrai si test1 ou test2 est vrai mais pas si les deux sont vrais simultanément
Not	Négation	If Not test	Inverse le résultat du test
AndAlso	Et logique	If (test1) AndAlso(test2)	Idem “et logique” mais test2 ne sera évalué que si test1 est vrai
OrElse	Ou logique	If (test1) OrElse (test2)	Idem “ou logique” mais test2 ne sera évalué que si test1 est faux

Il convient d’être prudent avec les opérateurs AndAlso et OrElse car l’expression que vous testez en second (test2 dans notre cas) pourra parfois ne pas être exécutée. Si cette deuxième expression modifie une variable, celle-ci ne sera modifiée que dans les cas suivants :

- premier test vrai dans le cas du AndAlso
- premier test faux dans le cas du OrElse.

7. Ordre d'évaluation des opérateurs

Lorsque plusieurs opérateurs sont combinés dans une expression, ils sont évalués dans un ordre bien précis. Les opérations arithmétiques sont exécutées en premier puis les opérations de comparaison et enfin les opérateurs logiques.

Les opérateurs arithmétiques ont entre eux également un ordre d'évaluation dans une expression. L'ordre d'évaluation est le suivant :

- Puissance(^)
- Negation (-)
- Multiplication et division (*, /)
- Division entière (\)
- Modulo (Mod)
- Addition et soustraction (+, -), concaténation de chaînes (+)
- Concaténation de chaîne (&).

Si un ordre d'évaluation différent est nécessaire dans votre expression, placez les portions à évaluer en priorité, entre parenthèses, comme dans l'expression suivante :

```
X= (z * 4) ^ (y * (a + 2))
```

Vous pouvez utiliser autant de niveaux de parenthèses que vous le souhaitez dans une expression. Il importe cependant que l'expression contienne autant de parenthèses fermantes que de parenthèses ouvrantes sinon le compilateur générera une erreur.

Les structures de contrôle

Les structures de contrôle permettent de modifier l'ordre d'exécution des instructions dans votre code. Deux types de structures sont disponibles :

- Les structures de décision : elles aiguilleront l'exécution de votre code en fonction des valeurs que pourra prendre une expression de test.
- Les structures de boucle : elles feront exécuter une portion de votre code un certain nombre de fois, jusqu'à ce qu'une condition soit remplie ou tant qu'une condition est remplie.

1. Structures de décision

Deux solutions sont possibles.

a. Structure If

Quatre syntaxes sont utilisables pour l'instruction If.

```
If condition then instruction
```

Si la condition est vraie alors l'instruction est exécutée ; dans ce cas, "condition" doit être une expression qui, une fois évaluée, doit fournir un boolean `true` ou `false`. Avec cette syntaxe, seule l'instruction située après le `then` sera exécutée si la condition est vraie.

Pour pouvoir exécuter plusieurs instructions en fonction d'une condition, la syntaxe à utiliser est :

```
If condition then
Instruction 1
...
Instruction n
End if
```


Dans ce cas, le groupe d'instructions situé entre le `then` et le `End If`, sera exécuté si la condition est vraie.

Vous pouvez également spécifier une ou plusieurs instructions qui, elles, seront exécutées si la condition est fausse.

```
If condition then
Instruction 1
...
Instruction n
  Else
Instruction 1
...
Instruction n
End if
```

Vous pouvez également imbriquer les conditions avec la syntaxe :

```
If condition1 then
Instruction 1
...
Instruction n
  ElseIf Condition 2 then
  Instruction 1
  ...
  Instruction n
  ElseIf Condition 3 then
  Instruction 1
  ...
```

```
  Instruction n
Else
  Instruction 1
  ...
  Instruction n
End if
```

Dans ce cas, on teste la première condition. Si elle est vraie alors le bloc de code correspondant est exécuté, sinon on teste la suivante et ainsi de suite. Si aucune condition n'est vérifiée, le bloc de code spécifié après le `else` est exécuté. L'instruction `else` n'est pas obligatoire dans cette structure. Dans ce cas, il se peut qu'aucune instruction ne soit exécutée si aucune des conditions n'est vraie.

b. Structure Select case

La structure `select case` permet un fonctionnement équivalent, mais offre une meilleure lisibilité du code. La syntaxe est la suivante :

```
Select case variable
  Case valeur1
  Bloc de code 1
  Case valeur2
  Bloc de code 2
  Case valeur3
  Bloc de code 3
  Case else
  Bloc de code 4
End select
```

La valeur de la variable est évaluée au début de la structure (par le `Select case`) puis la valeur obtenue est comparée avec la valeur spécifiée dans le premier `case` (valeur1).

Si les deux valeurs sont égales, alors le bloc de code 1 est exécuté et l'exécution de votre code se poursuit par l'instruction placée après le `End select`.

Sinon, la valeur obtenue est comparée avec la valeur du `case` suivant, s'il y a correspondance, le bloc de code est exécuté et ainsi de suite jusqu'au dernier `case`.

Si aucune valeur concordante n'est trouvée dans les différents `case`, alors le bloc de code spécifié dans le `case else` est exécuté.

La valeur à tester peut être contenue dans une variable, mais elle peut également être le résultat d'un calcul. Dans ce cas, le calcul n'est effectué qu'une seule fois au début du `select case`. Le type de la valeur testée peut être numérique ou chaîne de caractères. Le type de la variable testée doit bien sûr correspondre au type des valeurs dans les différents `case`.

Les valeurs à tester dans les différents `case` peuvent être regroupées comme dans l'exemple suivant :

```
Select case reponse
  Case "oui","OUI"
    console.writeline("reponse positive")
  Case "non","NON"
    Console.writeline("reponse negative")
  Case else
    Console.writeline("reponse de normand")
End select
```

Ou bien définies sous forme d'intervalle ou de comparaison :

```
Select case heure
  Case 8 to 12
    Console.writeline("Matin")
  Case 13 to 18
    Console.writeline("après-midi")
  Case 19 to 22
    Console.writeline("soir")
  Case is >=22
    Console.writeline("nuit")
  Case else
    Console.writeline("heure invalide")
```

```
End select
```

Il existe également trois instructions permettant d'obtenir le même résultat qu'avec un `if then else` ou un `select case` mais avec une seule instruction :

- L'instruction `iif` permet de remplacer un `if then else` avec la syntaxe suivante :

```
iif (condition, valeur renvoyée si vrai, valeur renvoyée si faux)
```

Exemple :

```
resultat = iif (reponse="oui", "YES", "NO")
```

- L'instruction `switch` permet d'indiquer un ensemble d'expressions et de valeurs associées et retourne la valeur associée à la première expression qui est vraie.

Exemple :

```
resultat=switch(reponse="oui","YES",reponse="non","NO")
```

- L'instructio nchoose permet de choisir une valeur dans la liste des paramètres, en fonction de la valeur d'un index. L'exemple suivant nous permet de choisir le taux de TVA en fonction du code (de 1 à 4).

Exemple :

```
tauxTva = Choose(code, 0, 5.5, 19.6, 33)
```

2. Les structures de boucle

Quatre structures sont à notre disposition :

```
While ... End While
Do ... Loop
For ... Next
For Each ... Next
```

Elles ont toutes pour but d'exécuter un bloc de code un certain nombre de fois en fonction d'une condition.

a. Structure While ... End While

```
While condition
  Bloc de code
End While
```

Cette syntaxe permet d'exécuter le bloc de code tant que la condition est vraie. La condition est évaluée avant même le premier passage dans la boucle, donc le bloc de code pourra très bien ne jamais être exécuté si la condition est fausse dès le départ. Dans le cas où la condition est vraie au premier passage, le bloc de code est exécuté ; la condition est à nouveau testée, si elle est vraie, une exécution du bloc de code est effectuée, sinon la prochaine instruction exécutée sera celle qui suit le `End While`. Il est toutefois possible de prévoir une sortie "prématurée" de la boucle en utilisant l'instruction `Exit While`. L'exécution reprend donc sur la ligne qui suit immédiatement le `End While`.

b. Structure Do ...Loop

La structure `Do Loop` nous propose quatre variantes :

```
Do While condition
  Bloc de code
Loop
```

Le fonctionnement de cette syntaxe est strictement identique à la structure `While End While`.

```
Do
  Bloc de code
Loop While condition
```

Cette syntaxe nous permet de garantir que le bloc de code sera exécuté au moins une fois puisque la condition sera testée à la fin du bloc de code.

Les instructions précédentes réalisent une boucle tant qu'une condition était remplie, alors que les deux syntaxes suivantes effectuent une boucle jusqu'à ce qu'une condition soit remplie.

```
Do until condition
  Bloc de code
Loop
```

Dans ce cas, la boucle est exécutée jusqu'à ce que la condition soit vraie. Si elle est vraie dès le début de la boucle, le bloc de code ne sera jamais exécuté. Pour garantir au moins une exécution du bloc de code, il convient d'utiliser la syntaxe suivante qui teste la condition à la fin de l'exécution du bloc de code.

```
Do
  Bloc de code
Loop until condition
```

Le bloc de code s'exécute au moins une fois puis teste la condition et boucle jusqu'à ce que la condition soit vraie. Comme dans le cas de la boucle while, l'instruction `Exit Do` provoquera une sortie anticipée et inconditionnelle de la boucle.

c. Structure For ... Next

Lorsque vous connaissez le nombre d'itérations à réaliser dans une boucle, il est préférable d'utiliser la structure `for ... next`. Pour pouvoir utiliser cette instruction, une variable de compteur doit être déclarée dans le code, pour votre boucle.

La syntaxe générale est la suivante :

```
For Compteur=valeur initiale To valeur Finale
  Bloc de code
Next
```

Au début de la boucle, la variable `Compteur` est initialisée avec la `valeur initiale` puis le bloc de code est exécuté. L'instruction `next` provoque l'incrémentation de la variable `Compteur` et la comparaison de la valeur obtenue avec la valeur `Finale` de la boucle. Si la variable `compteur` est inférieure ou égale à la valeur `Finale`, le bloc de code est exécuté à nouveau, sinon l'exécution se poursuit à l'instruction qui suit le `next`.

Par défaut, l'incrément est de un ; vous pouvez spécifier une valeur pour l'incrément en ajoutant le mot clé step et en spécifiant ensuite l'incrément. Cette valeur peut être négative mais dans ce cas, une `valeur` initiale supérieure à la valeur finale devra être indiquée.

Comme pour les autres structures, il est possible de sortir immédiatement d'une boucle `for next` en utilisant l'instruction `exit for`.

d. Structure For each ... next

Une autre syntaxe de la boucle `for next` permet d'exécuter un bloc de code pour chaque élément contenu dans un tableau ou dans une collection. La syntaxe générale de cette instruction est la suivante :

```
For each element in tableau
  Bloc de code
Next
```

Il n'y a pas de notion de compteur dans cette structure, puisqu'elle effectue elle-même les itérations sur tous les éléments présents dans le tableau ou la collection.

La variable element sert à extraire les éléments du tableau ou de la collection pour que le bloc puisse le manipuler. Le type de la variable element doit être compatible avec le type des éléments stockés dans le tableau ou la collection. Par contre, vous n'avez pas à vous soucier du nombre d'éléments car l'instruction for each est capable de gérer elle-même le déplacement dans le tableau ou la collection. Voici un petit exemple pour clarifier la situation !

Avec une boucle classique :

```
Dim tablo() As String = {"rouge", "vert", "bleu", "blanc"}
Dim cpt As Integer
For cpt = 0 To UBound(tablo)
    Console.WriteLine(tablo(cpt))
Next
```

Avec la boucle for each :

```
Dim tablo() As String = {"rouge", "vert", "bleu", "blanc"}
Dim s As String
For Each s In tablo
    Console.WriteLine(s)
Next
```

Comme pour la boucle for next, vous pouvez provoquer une sortie de la boucle avant d'avoir parcouru l'ensemble du tableau, en utilisant l'instruction exit for.

e. Autres structures

Deux autres structures plutôt destinées à simplifier le développement sont disponibles :

Structure Using End Using

Cette structure est destinée à accueillir un bloc de code utilisant une ressource externe, comme par exemple une connexion à un serveur de base de données. Cette structure prend en charge automatiquement la libération de la ressource à la fin du bloc de code. La ressource peut être créée dans la structure ou bien exister auparavant et être passée sous contrôle de la structure. À la fin de la structure, la ressource est libérée en appelant la méthode Dispose.

Exemple

```
Dim ctn As New .SqlConnection
ctn = New SqlConnection("Data Source=TG;Initial Catalog=Northwind;Integrated
Security=True")
Using ctn
...
' Utilisation de la connexion
...
End Using
' La connexion est libérée
```

Structure With End With

Cette structure permet d'exécuter une série d'opérations sur un objet sans avoir à rappeler à chaque fois son nom. Elle facilite énormément la lecture du code et améliore également les performances.

Il suffit de spécifier, à l'aide du mot clé With, le nom de la variable à utiliser et jusqu'au mot clé End With ; ce nom sera sous-entendu. Les différents éléments de l'objet seront disponibles sans les préfixer par le nom de la variable.

Exemple

```
Dim ctn As New SqlConnection
ctn = New SqlConnection("Data Source=TG;Initial Catalog=Northwind;Integrated
Security=True")
            With ctn
                .Open()
                '....
                '....
                .Close()
            End With
```

Les procédures et fonctions

Dans une application Visual Basic, toutes les instructions doivent obligatoirement être placées dans une procédure ou une fonction. Ces procédures ou fonctions nous permettent de créer des blocs de code qui pourront ensuite être appelés dans d'autres portions de votre application. L'appel à la procédure ou fonction se fera simplement en utilisant l'identifiant de la procédure.

Pour que ces procédures soient plus facilement réutilisables, vous avez la possibilité d'utiliser des paramètres. Les valeurs de ces paramètres seront spécifiées au moment de l'appel de la procédure.

Au cours du développement, n'hésitez pas à créer de nombreuses procédures et fonctions. Le découpage de votre application en de nombreuses procédures et fonctions facilitera le débogage (une dizaine de blocs de code d'une quinzaine de lignes est plus facile à tester qu'un "pavé" de cent cinquante lignes). Certaines procédures peuvent même être réutilisées plusieurs fois dans votre application.

Dans Visual Basic, quatre types de procédures sont disponibles.

- Les procédures Sub qui exécutent simplement un bloc de code à la demande.
- Les procédures événementielles qui sont appelées automatiquement lorsqu'un événement se produit pendant le fonctionnement de votre application (clic souris, touche clavier...).
- Les fonctions qui exécutent un bloc de code et renvoient le résultat de leur calcul au code qui les a appelées.
- Les procédures property qui permettent de manipuler les propriétés des objets créés dans l'application.
- Les procédures opérateur utilisées pour modifier le fonctionnement d'un opérateur lorsqu'il s'applique à une classe ou une structure.

Voyons maintenant comment déclarer des procédures et fonctions.

1. Procédure Sub

Le code d'une procédure doit être placé entre les mots clés `Sub` et `End Sub`. La procédure doit être nommée. Ce nom sera utilisé lors de l'appel. La syntaxe générale de déclaration est la suivante :

```
Sub AfficheResultat()
  Instruction 1
  ...
  console.System.Writeline( "ça marche !!! ")
  ...
  Instruction n
End Sub
```

Les parenthèses après le nom sont utilisées pour spécifier les caractéristiques des paramètres qui seront passés lors de l'appel. Les parenthèses sont obligatoires dans la déclaration même si aucun paramètre n'est requis pour la procédure.

De nombreux mots clés sont utilisables, dans la déclaration d'une procédure, pour modifier les possibilités de réutilisation de cette procédure. La plupart d'entre eux sont liés à la programmation objet et seront étudiés dans un autre chapitre. Par contre, pour modifier la visibilité de votre procédure, vous pouvez utiliser les mots clés que nous avons déjà utilisés pour la déclaration des variables (*Private Public Friend*). Sans spécification, une procédure sera considérée publique.

Pour demander l'exécution de votre procédure dans le code, il suffit de spécifier son nom. Une autre méthode utilisant le mot clé `Call` est aussi possible, la syntaxe est alors la suivante :

```
Call AfficheResultat()
```

Si votre procédure n'attend pas de paramètres, l'utilisation des parenthèses est optionnel (l'environnement de développement Vb les ajoutera cependant, automatiquement, dans votre code).

2. Procédure événementielle

Deux éléments distinguent une procédure classique d'une procédure événementielle :

- l'appel de la procédure se fait automatiquement lorsque l'événement géré par la procé- dure se produit dans votre application,
- le nom de la procédure correspond à la concaténation du nom de l'élément pour lequel elle gère l'événement, du caractère de soulignement (_) puis du nom de l'événement géré. Par exemple, la procédure suivante sera exécutée lorsque l'événement `click` se produira sur le bouton **BpOk**.

```
Sub BpOk_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles
Button1.Click
```

Le mot clé `Handles` spécifie l'objet et l'événement pour lesquels la procédure sera exécutée.

---

3. Fonction

Une fonction se déclare suivant le même principe qu'une procédure, en utilisant les mots clés `Function` et `End Function`. Vous devez, par contre, fournir une information supplémentaire. Puisque la fonction doit renvoyer au code appelant une valeur, vous devez préciser le type de la valeur qui sera renvoyée. La syntaxe de déclaration est donc la suivante :

```
Function calcul() As Integer
  Instruction 1
  ...
  ...
  Instruction n
End Function
```

Dans le code de votre fonction, vous allez devoir spécifier quelle valeur sera renvoyée par votre fonction. Pour cela, deux solutions sont possibles :

- Utilisation du mot clé `Return` en indiquant la valeur que vous voulez renvoyer par la fonction. Dans ce cas, `return` provoque immédiatement la sortie de la fonction, même si ce n'est pas la dernière instruction.
- Vous pouvez également, dans le code de la fonction, utiliser le nom de la fonction comme une variable et lui affecter la valeur de retour de la fonction. L'exécution de la fonction se poursuit dans ce cas jusqu'à la dernière instruction.

Une fonction peut ensuite être utilisée dans le code à la place d'une valeur du même type que celui renvoyé par la fonction. Elle peut également être utilisée comme une procédure `Sub` ; dans ce cas, la valeur renvoyée sera tout simplement ignorée.

4. Procédures Property

Les procédures `Property` vont nous permettre d'ajouter une propriété à une classe, un module ou une structure. Ces procédures sont parfois appelées "accesseurs". Elles seront utilisées lorsque l'on modifie (`Set`) ou que l'on utilise (`Get`) la propriété. Leur utilisation semble similaire à l'utilisation d'une variable, on peut affecter une valeur à une propriété ou lire la valeur d'une propriété. Cependant, il existe de nombreuses différences importantes entre les variables et les propriétés :

- Les variables ne nécessitent qu'une seule ligne de code pour la déclaration.
- Les propriétés nécessitent un bloc de code pour la déclaration.
- L'accès à une variable s'effectue directement.
- L'accès à une propriété implique l'exécution d'une portion de code.
- Le contenu d'une variable est toujours récupéré tel quel.

- Le contenu d'une propriété peut être modifié par le code, lors de l'accès à la propriété.

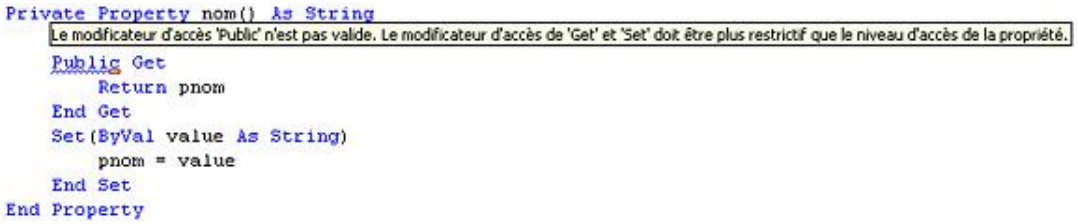
La syntaxe générale de création d'une propriété est la suivante :

```
Property nomPropriété () as typeDeLaPropriété
  Get
    ...
    Return ...
  End Get

  Set
    ...
  End Set (...)
End Property
```

Dans cette déclaration

- `nomPropriété` correspond au nom par lequel la propriété est manipulable dans le code.
- `TypeDeLaPropriété` correspond au type de données associé à la propriété. Vous pouvez utiliser n'importe quel type de données pour une propriété (les types de base du langage ou un type personnalisé comme par exemple une classe).
- Le bloc compris entre `Get` et `End Get` contient le code exécuté lors de la lecture de la propriété.
- Le bloc compris entre `Set` et `End Set` contient le code exécuté lors de l'affectation d'une valeur à la propriété. Le paramètre est destiné, comme dans le cas d'une procédure classique, à faire passer de l'information vers le bloc de code.

Comme pour tout élément déclaré dans Visual Basic, vous pouvez spécifier un modificateur de niveau d'accès pour une propriété. Il s'applique au bloc `Get` et `Set`. Vous pouvez également spécifier un modificateur de niveau d'accès pour chacun des blocs `Get` et `Set`. Dans ce cas, ils doivent être plus restrictifs que celui indiqué au niveau de la propriété.



Les propriétés peuvent également être en lecture seule ou en écriture seule. Vous devez, dans ce cas, ajouter `ReadOnly` ou `WriteOnly` devant le nom de la propriété, en éliminant bien sûr le bloc de code `Set` dans le cas d'un propriété en lecture seule, et le bloc `Get` dans le cas d'une propriété en écriture seule.

5. Les procédures opérateur

Ce type de procédure permet la redéfinition d'un opérateur<$I[] standard du langage pour l'utiliser sur des types personnalisés (classe ou structure). Prenons un exemple avec la structure client déjà utilisée.

```
Public Structure Client
  Public Code As Integer
  Public Nom As String
  Public Prenom As String
End Structure
```

Essayons le code suivant :

```
Public Sub test()
    Dim c1, c2, c3 As Client
    c1.Code = 200
    c1.Nom = "client1"
    c1.Prenom = "prenom1"
    c2.Code = 125
    c2.Nom = "client2"
    c2.Prenom = "prenom2"
    c3 = c1 + c2
    Consc
    Console.WriteLine(c3.Nom)
    Console.WriteLine(c3.Prenom)
End Sub
```

L'opérateur '+' n'est pas défini pour les types 'serializable.Compta.Facturation.test.client' et 'serializable.Compta.Facturation.test.client'.

Visiblement, le compilateur n'est pas coopératif à l'idée d'additionner deux clients.

Pour que ce code fonctionne, nous devons lui indiquer la procédure à suivre pour réaliser cette opération. Nous devons donc redéfinir l'opérateur "+" pour l'utiliser avec deux clients.

```
Public Structure Client
            Public Code As Integer
            Public Nom As String
            Public Prenom As String
            Public Shared Operator +(ByVal c1 As Client, ByVal c2 As
Client) As Client
                Dim c As Client
                c.Code = c1.Code + c2.Code
                c.Nom = c1.Nom & c2.Nom
                c.Prenom = c1.Prenom & c2.Prenom
                Return c
            End Operator
        End Structure
```

Après cette modification, le compilateur se montre plus coopératif et l'exécution de la procédure précédente test affiche le résultat suivant :

```
325
client1client2
prenom1prenom2
```

6. Les arguments des procédures et fonctions

Pour que le code soit plus facilement réutilisable, les valeurs qui sont manipulées par les procédures et fonctions peuvent être passées comme paramètres au moment de l'appel de la procédure ou fonction. Lors de la déclaration de la procédure, vous devez spécifier la liste des paramètres qui seront attendus. Cette liste est située entre les parenthèses de la déclaration de la procédure. Vous devez indiquer, pour chaque paramètre, son nom et son type. Si plusieurs paramètres sont attendus, il convient de les séparer par une virgule.

Dans le code de la procédure, les paramètres sont considérés comme des variables déclarées localement.

Lors de l'appel de la procédure, une valeur, pour chacun des paramètres attendus, devra être indiquée. Prenons un exemple de déclaration et d'utilisation :

```
Function CalculTTC(Pht as double, taux as double) as double
  CalculTTC = Pht * (1 + (taux / 100))
End function

PrixHt = 100
PrixTtc = CalculTTC (PrixHt,5.5)
Console .Writeline (PrixTtc)
```

Pour passer une variable comme paramètre à une procédure (le PrixHt de l'exemple précédent), il existe deux possibilités :

- le passage par valeur : dans ce cas, l'information transmise à la procédure sera simple- ment le contenu de la variable passée comme paramètre. Le passage par valeur est l'option par défaut dans Visual Basic.

- Le passage par référence : dans ce cas, l'information transmise à la procédure n'est plus le contenu de la

variable mais l'emplacement où est stockée la variable, dans la mémoire de la machine. Le code de la procédure va donc chercher à cet emplacement la valeur dont elle a besoin. Le code de la procédure peut également modifier le contenu de la variable et ,dans ce cas, les modifications seront visibles dans le code qui a appelé votre procédure. Pour utiliser ce mode de passage, vous devez utiliser le mot clé `ByRef` devant le nom du paramètre dans la déclaration de la procédure. Bien qu'il ne soit pas nécessaire, le mot clé `ByVal` est également disponible pour spécifier un passage par valeur.

Vous pouvez également indiquer, dans la liste des paramètres d'une procédure, que certains seront optionnels en plaçant le mot clé `optional` devant le nom du paramètre. Les paramètres optionnels doivent toutefois respecter certaines règles :

- Pour chaque paramètre optionnel, une valeur par défaut doit être spécifiée dans la déclaration de la procédure.

```
Function CalculTTC(Pht as double, Optional taux as double = 19.6)
as double
```

- Lorsqu'un paramètre est déclaré optionnel dans une procédure, tous les suivants doivent également être déclarés optionnels. La déclaration suivante est invalide car le troisième paramètre doit également être optionnel :

```
Function CalculTTC(Pht as double, Optional taux as double =
19.6,devise as string) as double
```

La syntaxe doit être la suivante :

```
Function CalculTTC(Pht as double, Optional taux as double =
19.6,optional devise as string = "Ç") as double
```

Lors de l'appel de la procédure, vous avez deux possibilités pour indiquer la valeur utilisée pour chaque paramètre.

- le passage par position : Les valeurs des paramètres doivent apparaître dans le même ordre que dans la déclaration de la procédure. Si vous voulez omettre une valeur pour un paramètre, sa place devra tout de même être réservée dans l'appel de la procédure.

```
Resultat=calculTTC(250,,"$")
```

- Le passage par nom : Vous devez dans ce cas, lors de l'appel de la procédure, indiquer le nom de chaque paramètre et la valeur que vous voulez lui affecter. L'ordre des paramètres n'a pas d'importance mais vous êtes quand même obligé de spécifier une valeur pour les paramètres non optionnels.

```
Resultat=calculTTC(devise := "$",Pht := 250)
```

Une autre possibilité permet de créer une procédure qui pourra prendre un nombre quelconque de paramètres. Indiquez le mot clé `ParamArray` pour déclarer un tableau de paramètre.

Dans l'exemple suivant, nous allons créer une fonction qui calcule la moyenne de tous les paramètres qui lui sont passés.

```
Function moyenne(ByVal ParamArray notes() As Double) As Double
        Dim note As Double
        Dim somme As Double
        For Each note In notes
            somme = somme + note
        Next
        moyenne = somme / notes.Length
    End Function
```

La fonction peut ensuite être appelée avec un nombre quelconque de paramètres.

```
Resultat=moyenne(1,6,23,45)
```

ou

```
Resultat=moyenne(12,78)
```

Assemblies, Namespace et attributs

1. Les assemblies

Visual Basic est conçu autour du framework .NET, ce qui lui permet de bénéficier de nombreux avantages notamment en termes de sécurité, lors de l'exécution et de la gestion de la mémoire. Cette imbrication permet également d'assurer la compatibilité entre du code écrit dans les différents langages disponibles. Vous pouvez ainsi utiliser dans Visual Basic des éléments conçus avec d'autres langages (et inversement), de manière totalement transparente sans même vous soucier du langage dans lequel l'élément a été développé.

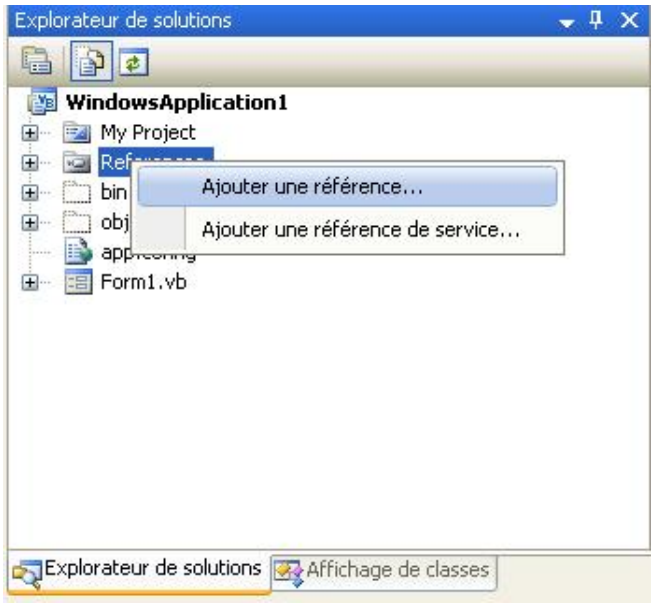
L'élément de base de cette réutilisation dans le framework .NET est l'assembly. Il peut être considéré comme le regroupement de types, de ressources et de fonctionnalités conçus pour fonctionner ensemble.

Les assemblies sont stockés dans des fichiers .exe ou .dll en fonction du type. Ils sont simplement générés par la compilation du projet correspondant.

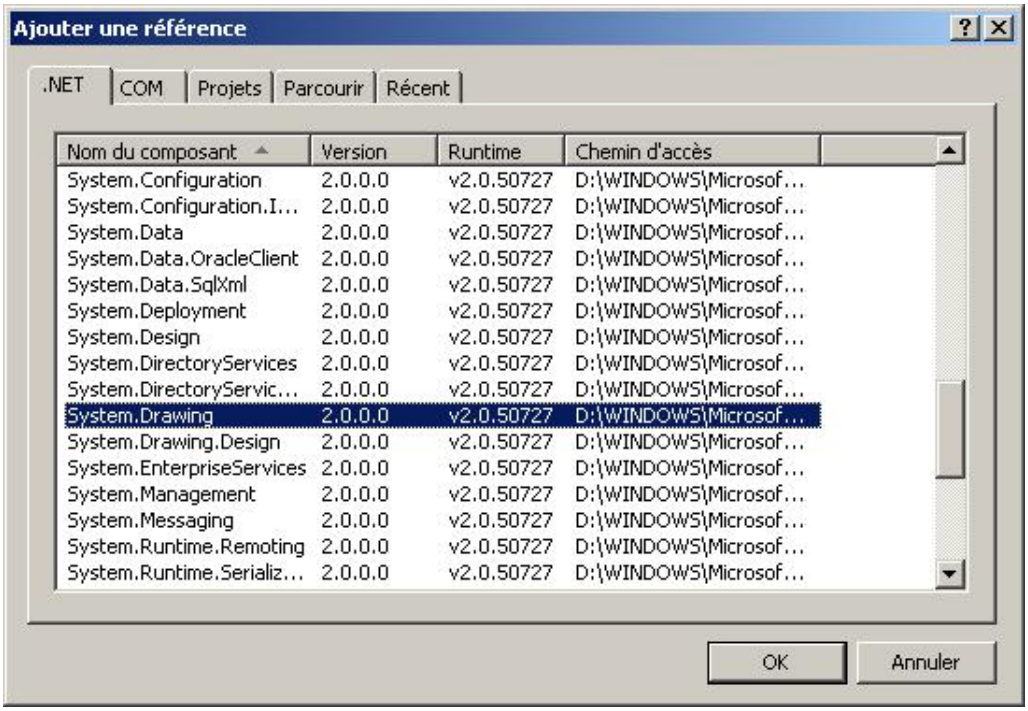
Ils sont autodescriptifs car ils contiennent les informations nécessaires pour leur utilisation dans un autre projet. Ces informations sont contenues dans le`manifest` de l'assembly. Le `manifest` contient entre autres :

- l'identité de l'assembly (son nom et sa version)
- une liste des fichiers utilisés par l'assembly (par exemple les autres assemblies utilisés par celui-ci, les ressources bitmap, etc.).

Pour pouvoir utiliser un assembly dans un projet, ajoutez simplement une référence vers l'assembly. Pour cela, utilisez le menu contextuel du dossier référence du projet.



La boîte de dialogue suivante permet alors de choisir les références à ajouter au projet.



Les différents onglets permettent de choisir, par catégorie, le type de référence à ajouter au projet :

.NET

L'ensemble des composants du Framework .NET disponibles.

COM

Les composants COM et ActiveX enregistrés sur le système.

Projects

Les autres projets de la solution courante.

Browse

Recherche d'un fichier (dll, ocx...) contenant les ressources.

Il est possible d'ajouter plusieurs références simultanément, en utilisant la touche [Ctrl] lors de la sélection dans cette boîte de dialogue.

Après avoir réalisé ces deux opérations, les ressources présentes dans l'assembly sont directement accessibles dans le code du projet.

2. Les Namespaces

Les namespaces organisent logiquement les objets disponibles dans un assembly. Ils sont utilisés pour lever les ambiguités lorsque, dans un projet, des références sont ajoutées sur des assemblies contenant des éléments ayant des noms identiques.

Par exemple, la classe `ListBox` existe dans les assemblies `System.Web` et `System.Windows.Forms`. Si des références sont ajoutées dans un projet vers ces deux assemblies, le compilateur risque de ne pas pouvoir déterminer laquelle de ces classes vous souhaitez réellement utiliser.

L'utilisation du nom pleinement qualifié, incluant le namespace dans lequel la classe est définie, permet de résoudre ce genre de problème.

Vous pouvez par exemple utiliser le code suivant :

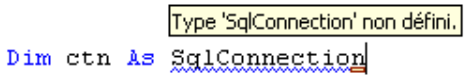
Exemple

```
Dim listeWindows As System.Windows.Forms.ListBox
```

```
Dim listeWeb As System.Web.UI.WebControls.ListBox.
```

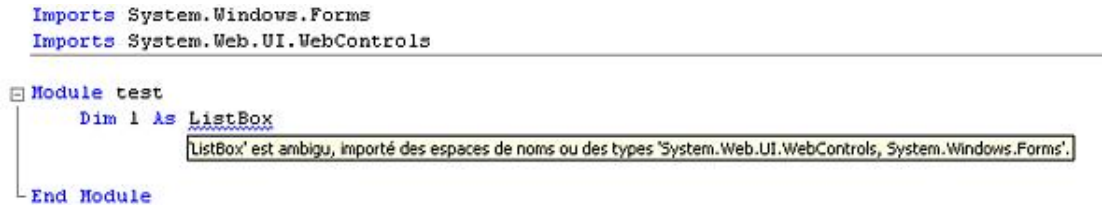
Cependant, l'utilisation du nom pleinement qualifié peut devenir pesant dans l'écriture du code. Il est possible d'utiliser le mot clé `imports` pour alléger le code. Il indique au compilateur que certains namespaces sont sous-entendus.

Par exemple l'instruction `Imports System.Data.SqlClient` autorise l'utilisation de la déclaration suivante : `Dim ctn As SqlConnection` qui sans importation du name- space aurait provoqué une erreur de compilation :



Les instructions `Imports` doivent être les premières lignes de code d'un fichier source vb.

Cependant, soyez vigilant pour ne pas retomber sur le problème précédent.



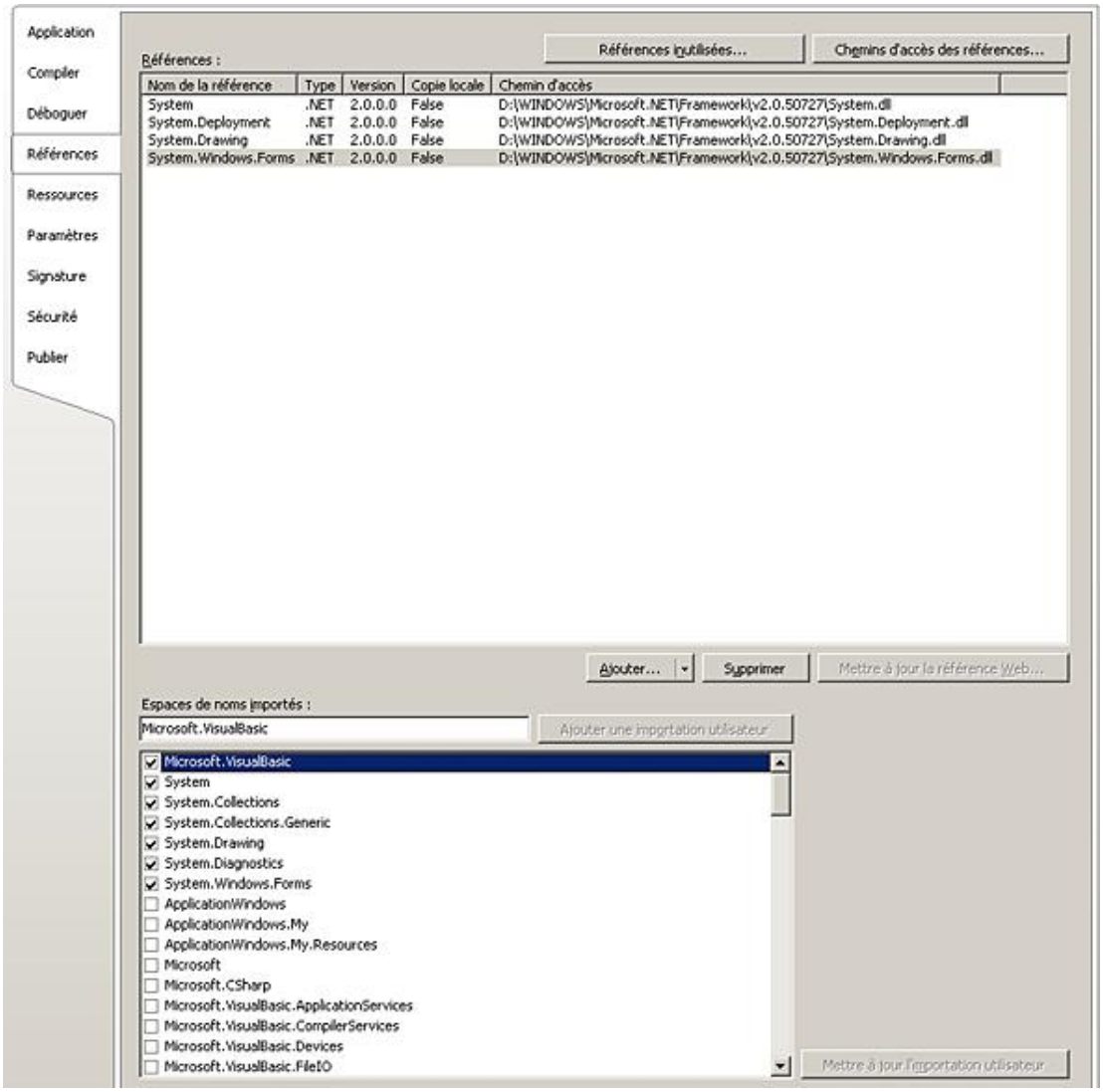
L'instruction `Imports` propose une solution élégante en créant un alias lors de l'importation du namespace.

```
Imports ctrlWin = System.Windows.Forms
Imports ctrlWeb = System.Web.UI.WebControls
Module TestCompta
    Dim listeWindows As ctrlWin.ListBox
    Dim listeWeb As ctrlWeb.ListBox
```

Cette solution autorise l'utilisation de noms d'une longueur raisonnable en évitant les conflits.

Il est également à noter que, en fonction du type de projet sur lequel vous travaillez, des références et des importations sont réalisées par défaut.

Vous pouvez vérifier ces références par défaut et, éventuellement les modifier, en affichant les propriétés de votre projet par le menu contextuel disponible dans l'explorateur de solutions.



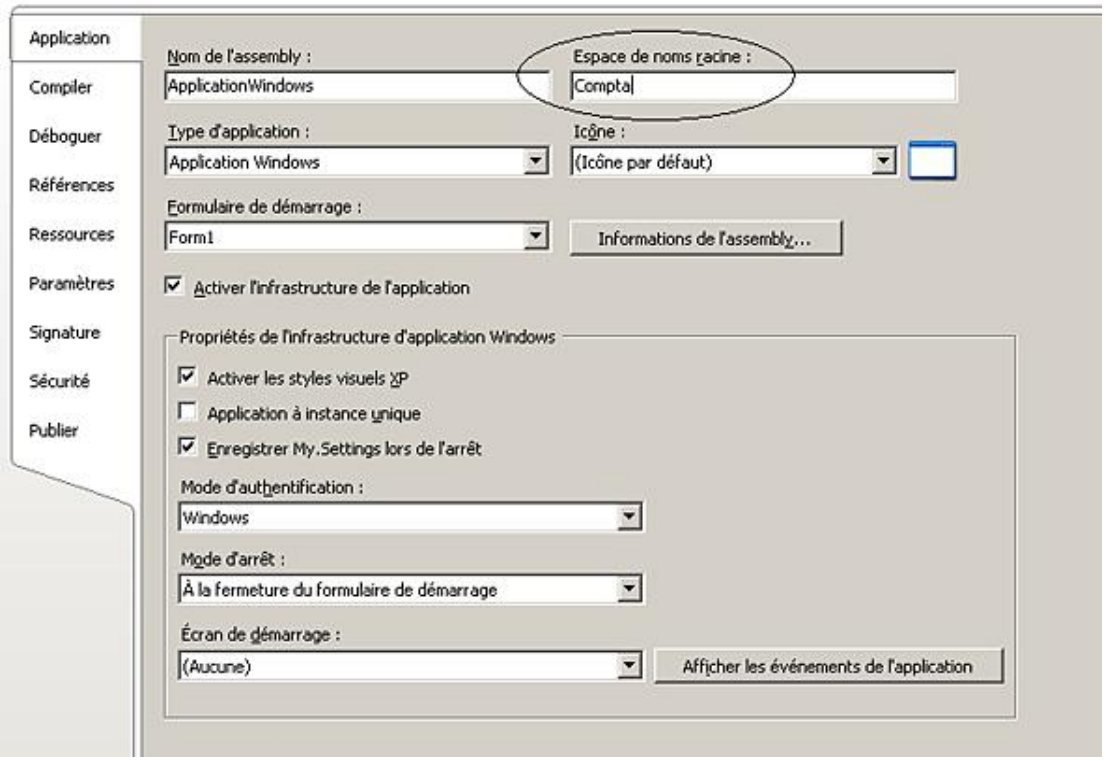
Les importations par défaut seront actives dans chaque module de votre projet.

Les namespaces seront déclarés dans le code à l'aide des mots clés Namespace nom du namespace et End Namespace.

Tous les éléments déclarés entre ces deux mots clés seront accessibles en les préfixant avec le nom du namespace.

```
Namespace Facturation
Public Class Tarification
  Public Shared Function calculTTC(prixHt As Decimal, tauxTva As Decimal)
As Decimal
     Return prixHt * (1 + tauxTva / 100)
  End Function
End Class
End Namespace
```

Dans l'exemple précédent, la fonction `calculTTC` définie dans la classe `Tarification` est accessible en la préfixant par le nom du namespace. Toutefois le nom du namespace racine spécifié devra également être ajouté au niveau des propriétés du projet, en le séparant des autres noms de namespace par un point.



Dans notre exemple, la fonction `calculTTC` est donc accessible par le code suivant :

```
Sub Main()
       prixTtc = Compta.Facturation.Tarification.calculTTC(100, 5.5)
End Sub
```

Utilisez la même technique, dans le cas de namespaces imbriqués ; comme dans l'exemple suivant :

```
Namespace Gestion
    Namespace Paye
        Public Class Salaire
            ....
        End Class
    End Namespace
    Namespace Facturation
        Public Class Facture
            ....
        End Class
    End Namespace

End Namespace
```

La classe `Salaire` sera donc accessible avec le nom `Compta.Gestion.Paye.Salaire`.

3. Les attributs

Les attributs sont des marques que vous pouvez placer dans votre code afin d'ajouter des informations supplémentaires aux éléments de votre application.

Ils sont sauvegardés dans les métadonnées de l'assembly pendant la compilation du projet. Les métadonnées sont utilisées par le runtime pour gérer le débogage, le suivi des versions, la compilation et d'autres informations sur l'utilisation de votre code. Les attributs peuvent s'appliquer à un assembly, un module ou une portion de code plus petite, telle qu'une procédure ou fonction. Ils pourront parfois accepter des arguments pour modifier leur signification.

Les attributs sont placés dans le code entre les symboles "<" et ">" comme une balise Html. Si plusieurs attributs sont utilisés, ils doivent être séparés par des virgules. Les éventuels paramètres d'un attribut seront placés entre parenthèses.

La portée d'un attribut peut également être étendue par les mots clé Assembly : ou Module : placés avant l'attribut. La syntaxe d'utilisation d'un attribut est donc :

```
<[Assembly ou Module :] Attribut1([paramètre1,paramètre2]),Attribut2
```

a. Attributs les plus courants en Visual Basic

Parmi les attributs disponibles, certains d'entres eux sont très fréquemment utilisés dans le développement avec Visual Basic. Nous allons étudier leur utilisation et l'illustrer par un exemple.

VBFixedStringAttribute:

Cet attribut force la création d'une variable de type chaîne de caractères de longueur fixe. Il est utilisé lors de la déclaration d'une structure, en vue de l'enregistrement dans un fichier pour avoir des enregistrements de taille constante.

Exemple

```
Structure Client
<VBFixedString(15)> public nom as String
<VBFixedString(15)> public prenom as String
End Structure
```

ComClassAttribute:

Cet attribut demande au compilateur la génération de code supplémentaire pour rendre compatible une classe avec le modèle COM. Cette classe sera ensuite compatible avec d'anciens langages de programmation incompatibles avec la plate-forme .NET.

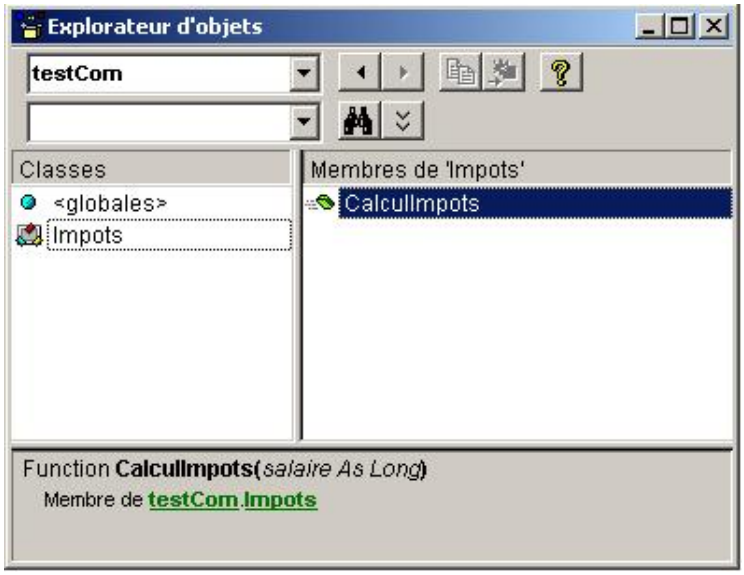
Exemple

```
<ComClass()> Public Class Impots

    Public Sub New()
        MyBase.New()
    End Sub

    Function CalculImpots(ByVal salaire As Integer)
        Return salaire * 0.3
    End Function
End Class
```

Pour compiler le code de cet exemple, il faut activer l'option au niveau des propriétés du projet dans l'option **Inscrire pour Com Interop**.

Après compilation, cette classe peut être utilisée dans un langage compatible COM, simplement en référençant la bibliothèque dll générée.

La figure suivante présente l'utilisation à partir de Vb 6.



SerializableAttribute, NonSerializedAttribute

Ces deux attributs contrôlent la sérialisation d'une classe et de ces membres. La sérialisation permet l'enregistrement d'une instance de classe, dans un fichier assurant ainsi la persistance des informations. Le fichier généré peut être au format binaire ou XML, dans ce cas, il facilite l'échange d'informations entre applications. Pour qu'une classe soit utilisable par le mécanisme de sérialisation, celle-ci doit être marquée avec l'attribut `SerializableAttribute`. Lors de l'opération de sérialisation, le contenu de chacun des membres de l'instance de la classe est enregistré dans le fichier. Si certains d'entre eux ne doivent pas être sauvegardés dans le fichier, ils doivent être marqués avec l'attribut `NonSerializedAttribute`.

L'exemple ci-après définit la classe `Personne` avec deux membres (Nom et Prenom) qui seront sérialisés et un membre (Age) qui ne sera pas sérialisé. Une instance de la classe est créée puis sauvegardée dans un fichier au format XML.

Exemple

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap

<Serializable()> Public Class Personne
    Public Nom As String
    Public Prenom As String
    <NonSerialized()> Public Age As Integer
    Public Sub New()
    End Sub
End Class
Module test
    Public Sub Main()
        Dim UnePersonne As New Personne()
        UnePersonne.Nom = "Dupond"
        UnePersonne.Prenom = "Paul"
        UnePersonne.Age = 25
        Dim stream As Stream = File.Open("donnees.xml", FileMode.Create)
        Dim formatter As New SoapFormatter()
        formatter.Serialize(stream, UnePersonne)
        stream.Close()
    End Sub
End Module
```

L'exécution de ce code génère le fichier XML suivant :

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.
org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/
" xmlns:clr=”http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-ENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Personne id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
serializable/serializable%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20
PublicKeyToken%3Dnull">
<Nom id="ref-3">Dupond</Nom>
<Prenom id="ref-4">Paul</Prenom>
</a1:Personne>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

On retrouve, sauvegardée dans ce fichier, notre instance de la classe `Personne` avec ses deux membres `Nom` et `Prenom` et, comme nous l'avons indiqué dans la définition de la classe, le membre `Age` n'est pas sauvegardé.

DllImportAttribute

Cet attribut est utilisé pour indiquer qu'une fonction est importée à partir d'une bibliothèque de code non managé. Il permet notamment l'utilisation de fonctions définies dans une bibliothèque du système. Dans l'exemple suivant, la fonction `MoveFile` peut être utilisée comme une fonction classique.

Exemple

```
<DllImport("KERNEL32.DLL")>Public Function  MoveFile(ByVal src As String,
ByVal dst As String) As Boolean
End Function
```

Introduction

Avec Visual Basic.NET, la notion d'objet est devenue omniprésente et nécessite un minimum d'apprentissage. Nous allons donc voir dans un premier temps le principe de la programmation objet et le vocabulaire associé, puis nous verrons comment mettre cela en application avec Visual Basic.

Dans un langage procédural classique, le fonctionnement d'une application est réglé par une succession d'appels aux différentes procédures et fonctions disponibles dans le code. Il n'y a aucune liaison entre les données et les procédures qui les manipulent. Dans un langage objet, on va au contraire essayer de regrouper au maximum les données et le code pour les manipuler. Les classes sont la représentation symbolique des objets. Elles décrivent les champs, propriétés, méthodes et événements de la même manière qu'un plan d'architecte décrit les différentes parties d'un bâtiment.

Poursuivons notre analogie entre une classe et un plan de bâtiment. Nous savons qu'Il est possible de construire plusieurs bâtiments à partir du même plan. De la même manière, plusieurs objets peuvent être construits à partir de la même classe. Une classe peut donc être utilisée pour créer autant d'instances que nécessaire.

Sur un plan de bâtiment, certaines zones peuvent avoir un accès limité à certaines personnes. De la même façon, dans une classe, certains éléments peuvent avoir un accès restreint. C'est le principe d'encapsulation.

Les termes classe et objet sont souvent confondus mais il s'agit, en fait, d'éléments bien distincts. Une classe représente la structure d'un élément alors que l'objet est un exemplaire créé sur le modèle de cette structure. La modification d'un élément dans un objet ne change absolument pas les autres objets créés à partir du même modèle (classe). Dans notre exemple de plan de bâtiment, l'ajout d'une nouvelle pièce à un bâtiment existant ne change rien aux autres bâtiments construits suivant le même plan. Par contre, la modification du plan (de la classe) entraîne des modifications pour tous les nouveaux bâtiments (tous les nouveaux objets).

Les classes sont constituées de champs, propriétés, méthodes et événements. Les champs et les propriétés représentent les informations contenues dans les objets. Les champs sont considérés comme des variables et il est possible de lire leur contenu ou de leur affecter une valeur directement. Par exemple, si vous avez une classe représentant un client, vous pouvez enregistrer son nom dans un champ.

Les propriétés se manipulent de la même façon que les champs, mais sont mises en œuvre à partir de procédures de propriété `Get` et `Set`. Ceci autorise plus de contrôle sur la façon dont les valeurs sont lues ou affectées et permet de valider les données avant leur utilisation.

Les méthodes représentent les actions qu'un objet peut effectuer. Elles sont mises en œuvre par la création de procédures ou fonctions dans une classe.

Les événements sont des informations qu'un objet reçoit ou transmet depuis ou vers un autre objet ou application. Les événements permettent aux objets d'exécuter des actions lorsqu'une situation particulière se produit. Comme Windows est un système d'exploitation événementiel, les événements peuvent provenir d'autres objets, du système ou des actions de l'utilisateur sur la souris et le clavier.

Ceci n'est qu'une facette de la programmation orientée objet. Trois autres éléments sont également fondamentaux :

- L'encapsulation.
- L'héritage.
- Le polymorphisme.

L'encapsulation est la capacité permettant de créer et de contrôler l'accès à un groupe d'éléments. Les classes fournissent le moyen le plus fiable d'assurer l'encapsulation. Si nous prennons l'exemple d'un compte bancaire, dans une programmation classique, il nous faudrait de nombreuses variables et procédures ou fonctions pour manipuler les informations. La situation serait encore plus complexe si nous devions gérer simultanément plusieurs comptes bancaire. Il faudrait alors travailler avec des tableaux et jongler avec les index. L'encapsulation permet de regrouper les informations et le code les manipulant dans une classe. Si vous devez travailler avec plusieurs comptes bancaire simultanément vous aurez alors plusieurs instances de la même classe, limitant ainsi le risque d'erreurs. L'encapsulation assure également un contrôle sur l'utilisation des données et des procédures ou fonctions. Vous pouvez utiliser les modificateurs d'accès, tels que `Private` ou `Protected`, pour restreindre l'accès à certaines méthodes, propriétés ou champs. Une règle fondamentale de l'encapsulation stipule que les données d'une classe ne doivent être manipulées que par le code de la classe (procédures de propriétés ou méthodes). Cette technique est parfois appelée dissimulation de données. Elle assure la sécurité de fonctionnement de votre code en masquant les détails internes de la classe et en évitant ainsi qu'ils ne soient utilisés de manière inappropriée. Elle autorise aussi la modification d'une partie du code sans perturber le fonctionnement du reste de l'application.

L'héritage permet la création d'une nouvelle classe, basée sur une classe existante. La classe servant de modèle pour la création d'une autre classe est appelée classe de base. La classe ainsi créée hérite des champs, propriétés, méthodes et événements de la classe de base. La nouvelle classe peut être personnalisée en y ajoutant des champs, propriétés, méthodes et événements. Les classes créées à partir d'une classe de base sont appelées classes dérivées. Vous pouvez donc définir une classe de base et la réutiliser plusieurs fois pour créer des classes dérivées.

Le polymorphisme est une autre notion importante de la programmation orientée objet. Par son intermédiaire, il est

possible d'utiliser plusieurs classes de manière interchangeable même si ces classes implémentent leurs propriétés et méthodes de manière différente. Ces propriétés et méthodes sont utilisables par le même nom, indépendamment de la classe à partir de laquelle l'objet a été construit.

Trois autres concepts sont également associés au polymorphisme. La surcharge, la substitution et le masquage de membres permettent la définition de membres d'une classe portant le même nom. Il existe cependant quelques petites distinctions entre ces trois techniques.

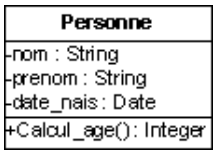
La surcharge est utilisée pour concevoir des propriétés ou des méthodes portant le même nom mais ayant un nombre de paramètres différents ou des types de paramètres différents.

La substitution permet la redéfinition de méthodes ou propriétés héritées d'une classe de base. Les membres substitués peuvent accepter le même nombre et type de paramètres que la méthode ou propriété de la classe de base.

Le masquage sert à remplacer localement, dans une classe, un membre d'une classe. N'importe quel type de membre peut masquer un autre membre. Par exemple, une propriété peut masquer une méthode héritée. Le masquage se fait uniquement grâce au nom. Les membres masqués ne sont pas héritables.

Mise en œuvre avec Visual Basic

Dans le reste de ce chapitre, nous allons travailler sur la classe `Personne` dont la représentation UML (*Unified Modeling Language*) est disponible ci-dessous.



UML est un langage graphique dédié à la représentation des concepts de programmation orienté objet. Pour plus d'informations sur ce langage, vous pouvez consulter l'ouvrage UML2 dans la même collection.

1. Création d'une classe

La création d'une classe passe par la déclaration de la classe elle-même et de tous les éléments la constituant.

a. Déclaration de la classe

La déclaration d'une classe se fait en utilisant les mots clés `Class` et `End Class`. Entre ces deux mots clés, on trouve des déclarations de variables qui seront les champs de la classe et des procédures qui seront les méthodes de la classe.

La syntaxe générale de définition d'une classe est donc la suivante :

```
[ Public | Private | Protected | Friend | Protected Friend ]
[ MustInherit | NotInheritable ]
Class nom de la classe
[ Inherits nom de la classe de base]
[ Implements nom de l'interface]

End Class
```

De nombreux mots clés sont disponibles pour la personnalisation d'une classe. Au moment de sa déclaration, on peut spécifier la visibilité de la classe. Les mots clés suivants sont disponibles :

`Public`

La classe pourra être utilisée dans tout votre projet mais aussi dans d'autres projets.

`Friend`

L'accès à la classe est limité au projet dans lequel elle est définie.

`Private`

La classe ne peut être utilisée que dans le module dans lequel elle est définie.

`Protected`

La classe ne peut être utilisée que dans une sous-classe de celle dans laquelle elle est définie. Ce mot clé ne peut être utilisé que pour une classe déclarée dans une autre classe.

`Protected Friend`

Identique à l'union des portées `protected` et `friend`.

Vous pouvez également indiquer comment votre classe va se comporter vis-à-vis de l'héritage. Deux options sont possibles :

`MustInherit`

Indique que la classe sert de classe de base dans une relation d'héritage. Vous ne pourrez pas créer d'instances de cette classe. En général, dans ce genre de classe, seules les déclarations des méthodes sont définies, il faudra dans les classes dérivées écrire le contenu de ces méthodes.

`NotInheritable`

Cette classe sera la dernière de la hiérarchie. Il ne sera donc pas possible d'utiliser cette classe comme super classe d'une autre classe.

Pour indiquer que votre classe récupère les caractéristiques d'une autre classe par une relation d'héritage, vous devez utiliser le mot clé`Inherits` suivi du nom de la classe de base. Vous pouvez également implémenter dans votre classe une ou plusieurs interfaces. Ces deux notions seront vues en détail plus loin dans ce chapitre.

Le début de la déclaration de notre classe `Personne` est donc le suivant :

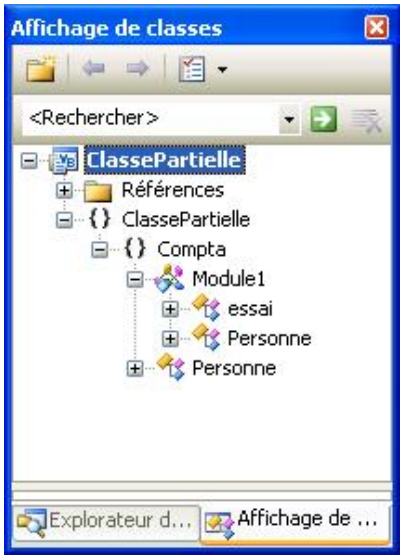
```
Public Class Personne
   Dim  nom As String
   Dim prenom As String
   Dim date_naiss As Date
End Class
```

b. Classe partielle

La définition d'une classe peut être répartie sur plusieurs déclarations en utilisant le mot clé `Partial`. Cette technique autorise la définition de la classe dans plusieurs fichiers sources. Elle est très utilisée dans Visual Studio pour permettre la personnalisation de classes générées automatiquement. Le code généré est en général placé dans un fichier nommé `.designer.vb` qui ne doit, en principe, pas être modifié directement. Lors de la compilation, le compilateur regroupe toutes les définitions partielles pour obtenir le code source de la classe. Les différentes parties de la définition d'une classe doivent par contre être dans le même projet et faire partie du même namespace. Il y a un autre petit piège à éviter également. Regardons le code suivant :

```
Namespace Compta
    Partial Public Class Personne
        Public Sub procedure1()
        End Sub
    End Class
    Module Module1
        Partial Public Class Personne
            Public Sub Procedure2()
            End Sub
        End Class
    End Module
End Namespace
```

Au premier abord rien d'illégal puisque le compilateur génère le code correctement. Par contre, il n'a pas la même vision des choses que nous. Regardons ce que nous présente l'explorateur de classes.



Deux classes Personne sont disponibles. Le compilateur a en fait considéré que nos deux définitions de classe ne font pas partie du même namespace. En effet l'une d'entre elle est située dans le module, et le nom de celui-ci est rajouté automatiquement comme préfixe à la classe.

c. Création de propriétés

Vous pouvez créer des variables simples pour stocker les informations de votre classe mais les procédures de propriété fournissent d'avantage de flexibilité et de contrôle sur le stockage des informations dans une classe. Elles permettent à la classe de protéger et valider ses propres données. Une procédure de propriété est définie entre les mots clés Property et End Property. Entre ces deux mots clés, deux blocs de code sont définis à l'aide des mots clés Get, End Get et Set End Set ; le bloc de code Get est exécuté lors de la lecture de la propriété, le bloc de code Set est exécuté lors de l'affectation d'une valeur à la propriété.

Notre classe Personne peut être améliorée de la façon suivante :

```
Public Class Personne
  Private leNom As String
 Private lePrenom As String
Private laDate_naiss As Date
   Public Property nom() As String
        Get
            Return leNom
        End Get
        Set(ByVal Value As String)
            leNom = Value
        End Set
    End Property

   Public Property prenom() As String
        Get
            Return lePrenom
        End Get
        Set(ByVal Value As String)
            lePrenom = Value
        End Set
    End Property
   Public Property date_naiss() As Date
        Get
            Return laDate_naiss
        End Get
        Set(ByVal Value As Date)
                LaDate_naiss = Value
        End Set
    End Property
End Class
```

La création des propriétés permet maintenant d'accéder de manière indirecte aux champs de la classe. Nous pouvons donc nous permettre de modifier la visibilité des champs de la classe et les rendre privés. C'est d'ailleurs une pratique recommandée pour respecter le principe d'encapsulation. Nous avons donc la possibilité d'être plus exigeant concernant les informations enregistrées dans notre classe. Nous allons mettre en pratique les quelques règles de gestion suivantes :

- Le nom sera stocké en majuscules.
- Le prénom sera stocké en minuscules.
- La date de naissance ne sera pas inférieure à 1900.

Les procédures de propriété sont donc chargées de l'application de ces règles.

```
    Property nom() As String
        Get
            Return leNom
        End Get

        Set(ByVal Value As String)
            leNom = UCase(Value)
```

```
        End Set
    End Property

    Property prenom() As String
        Get
            Return lePrenom
        End Get

        Set(ByVal Value As String)
            lePrenom = LCase(Value)
        End Set
    End Property

    Property date_naiss() As Date
        Get
            Return laDate_naiss
        End Get

        Set(ByVal Value As Date)
            If Value.Year >= 1900 Then
                laDate = Value
            End If
        End Set
    End Property
```

À noter que les procédures de propriété ont un accès complet aux champs de la classe, y compris ceux déclarés privés.

Lecture seule et écriture seule

Il peut parfois être intéressant de restreindre les accès possibles à une propriété. Elles peuvent donc être définies en lecture seule ou en écriture seule. Pour indiquer votre intention, vous devez utiliser les mots clés`ReadOnly` ou`WriteOnly`, lors de la déclaration de la propriété.

Le bloc de code `Get` doit être omis pour une propriété en écriture seule. Pour une propriété en lecture seule, c'est le bloc de code `Set` qui doit être omis. Pour mettre cela en application, nous allons ajouter à la classe `Personne` une propriété `MotDePasse` en écriture seule et une propriété `Age` en lecture seule. L'âge peut se déduire directement de la date de naissance et le mot de passe n'a pas à être accessible de l'extérieur de la classe.

```
    ReadOnly Property age() As Long
        Get
            Return DateDiff(DateInterval.Year, date_naiss, Now())
        End Get
    End Property
WriteOnly Property MotDePasse() as String
    Set (ByVal Value As String)
    LeMotDePasse=value
    End Set
End Property
```

Propriété par défaut

La propriété par défaut d'une classe est la propriété qui est utilisée lorsqu'aucune propriété particulière n'est indiquée pour un objet. C'est en général la propriété la plus utilisée d'une classe. Elle permet de rendre le code plus concis en évitant la répétition fréquente du même nom de propriété.

La propriété par défaut doit cependant respecter certaines contraintes :

- Elle doit accepter au moins un argument.
- Elle peut ne pas être privée ou partagée.
- Si une propriété surchargée est la propriété par défaut alors toutes les autres propriétés surchargées ayant le même nom doivent également être des propriétés par défaut.
- Il ne peut y avoir qu'une seule propriété par défaut dans une classe. Cette règle s'applique également pour

les propriétés héritées.

- Si une propriété par défaut d'une classe de base est masquée par une propriété qui n'est la propriété par défaut d'une sous-classe, alors la propriété par défaut de la classe de base est toujours accessible en n'utilisant que le nom de l'objet.

Mettons cela en application en ajoutant, à la classe `Personne`, la liste des enfants de cette personne et en définissant cette propriété comme propriété par défaut.

Le code de notre classe `Personne` devient donc :

```
Public Class Personne
    Private leNom As String
    Private lePrenom As String
    Private laDate_naiss As Date
    Private leMotDePasse As String
    Private lesEnfants(9) As Personne
    Property nom() As String
        Get
            Return leNom
        End Get

        Set(ByVal Value As String)
            leNom = UCase(Value)
        End Set
    End Property

    Property prenom() As String
        Get
            Return lePrenom
        End Get

        Set(ByVal Value As String)
            lePrenom = LCase(Value)
        End Set
    End Property

    Property date_naiss() As Date
        Get
            Return laDate_naiss
        End Get

        Set(ByVal Value As Date)
            If Value.Year >= 1900 Then
                laDate_naiss = (Value)
            End If
        End Set
    End Property
    ReadOnly Property age() As Long
        Get
            Return DateDiff(DateInterval.Year, date_naiss, Now())
        End Get
    End Property
    WriteOnly Property MotDePasse() As String
        Set(ByVal Value As String)
            LeMotDePasse = value
        End Set

    End Property
    Default Public Property enfant(ByVal index As Integer) As Personne
        Get
            Return lesEnfants(index)
        End Get
        Set(ByVal value As Personne)
            lesEnfants(index) = value
        End Set
    End Property
End Class
```

À noter que nous sommes obligés de créer un nouveau champ dans la classe `Personne` afin d'assurer le stockage de la liste des enfants. Pour l'instant, ce champ est constitué d'un tableau de personne mais pourra être avantageusement remplacé par une structure plus souple à gérer, comme par exemple une collection. La propriété `enfant` attend donc en paramètre un index permettant de spécifier l'enfant sur lequel nous souhaitons travailler.

Le code suivant nous permet de tester le bon fonctionnement de notre classe :

```
Module Principal
    Sub Main()
        Dim p As New Personne
        Dim enfant1 As New Personne
        Dim enfant2 As New Personne
        p.nom = "dupond"
        p.prenom = "paul"
        p.date_naiss = #12/23/1954#
        enfant1.nom = "dupond"
        enfant1.prenom = "pascal"
        enfant1.date_naiss = #10/5/1979#
        ' nous pouvons également utiliser le nom du parent pour
        ' initialiser le nom de l'enfant
        enfant2.nom = p.nom
        enfant2.prenom = "marc"
        enfant2.date_naiss = #4/18/1982#
        ' nous pouvons affecter un enfant à une personne en utilisant
        ' la propriété enfant
        p.enfant(0) = enfant1
        ' comme c'est également la propriété par défaut de la classe
        ' nous pouvons utiliser cette syntaxe
        p(1) = enfant2
        ' vérifions que nos informations sont correctes
        Console.WriteLine("Mr {0} {1} né le {2} a 2 enfants", p.nom, p.prenom,
p.date_naiss)
        Console.WriteLine("{0} {1} qui a {2} ans", p(0).nom, p(0).prenom,
p(0).age)
        Console.WriteLine("{0} {1} qui a {2} ans", p(1).nom, p(1).prenom,
p(1).age)

        Console.WriteLine("appuyer sur une touche pour quitter")
        Console.ReadLine()
    End Sub
End Module
```

Nous obtenons sur la console le résultat suivant :

Mr DUPOND paul né le 23/12/1954 00:00:00 a 2 enfants

DUPOND pascal qui a 26 ans

DUPOND marc qui a 23 ans

Appuyer sur une touche pour quitter.

Nous pouvons en profiter pour vérifier que nos règles concernant le nom et le prénom sont bien prises en compte : le nom est en majuscules, le prénom est en minuscules.

---

d. Création de méthodes

Les méthodes sont des procédures ou fonctions définies à l'intérieur d'une classe. Elles sont en général utilisées pour manipuler les champs et les propriétés de la classe. Pour pouvoir utiliser une méthode, il suffit de préfixer le nom de la méthode par le nom de l'objet sur lequel vous voulez appeler la méthode.

Ajoutons à la classe `Personne`, la fonction `Calcul_age()` et la procédure `Affichage` en insérant le code suivant :

```
    Public Function Calcul_age() As Integer
        Return DateDiff(DateInterval.Year, date_naiss, Now())
    End Function
    Public Sub affichage()
        Console.Write("Mr {0} {1} né le {2}", leNom, lePrenom, date_naiss)
    End Sub
```

À noter que dans ces lignes de code, nous pouvons manipuler les champs de la classe même s'ils sont déclarés privés, car nous sommes à l'intérieur de la classe. Il est également possible d'accéder aux informations de la classe en utilisant les propriétés. Dans ce cas, les règles de gestion concernant le nom et le prénom seront appliquées.

Nous pouvons modifier notre code de test pour utiliser la procédure et la fonction ajoutées à la classe.

```
p.affichage()
Console.WriteLine(" a 2 enfants")
Console.WriteLine("{0} {1} qui a {2} ans", p(0).nom, p(0).prenom, p(0).Calcul_age)
Console.WriteLine("{0} {1} qui a {2} ans", p(1).nom, p(1).prenom, p(1).Calcul_age)
Console.WriteLine("appuyer sur une touche pour quitter")
Console.ReadLine()
```

Surcharge de méthode

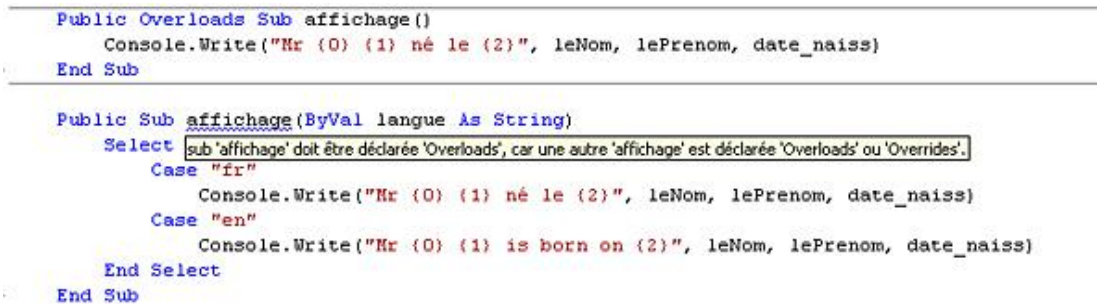
La surcharge de méthode est la création, au sein d'une classe, de méthodes ayant un nom identique mais un nombre de paramètres ou des types de paramètres différents. Ceci nous permet de conserver un nom cohérent pour plusieurs méthodes et dont le but est similaire mais pour lesquelles, seuls quelques détails changent. Les paramètres suivants ne sont pas pris en compte pour distinguer deux méthodes surchargées :

- Le nom des paramètres.
- Le type de retour d'une fonction.
- Les modificateurs `ByVal` ou `ByRef` appliqués aux paramètres de la méthode.

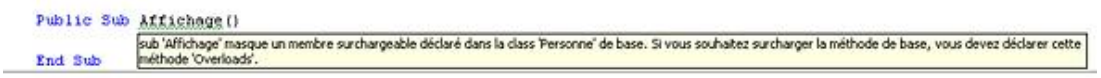
Nous pouvons, par exemple, surcharger la méthode `affichage` de la classe `Personne` pour prendre en compte la langue dans laquelle doit se faire l'affichage. Le paramètre attendu par la procédure permet de choisir la langue.

```
Public overloads  Sub affichage(ByVal langue As String)
 Select Case langue
   Case "fr"
   Console.Write("Mr {0} {1} né le {2}", leNom, lePrenom, date_naiss)
   Case "en"
   Console.Write("Mr {0} {1} was born the {2}", leNom, lePrenom, date_naiss)
 End Select
End Sub
```

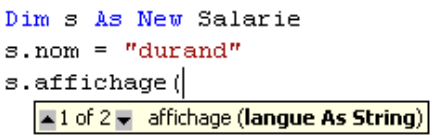
Le mot clé`overloads` est optionnel lors de la surcharge d'une méthode à l'intérieur d'une classe. Cependant, s'il est spécifié pour une méthode surchargée, il doit l'être pour toutes les autres.



Les méthodes héritées d'autres classes peuvent également être surchargées. Il convient cependant d'être prudent, dans ce cas, avec l'utilisation ou non du mot clé`overloads`. S'il n'est pas utilisé pour la surcharge alors toutes les autres surcharges de la classe de base seront inaccessibles. Par contre, s'il est utilisé, toutes les méthodes surchargées seront disponibles. Pour illustrer cela, nous allons considérer que nous avons une classe `salarie` qui hérite de la classe `Personne`. Nous verrons un petit peu plus loin comment mettre cet héritage en œuvre. Nous pouvons, dans notre classe `Salarie`, surcharger la méthode `Affichage()`. Si nous n'utilisons pas le mot clé `overloads`, Visual Studio nous prévient d'un problème potentiel, lié à l'inaccessibilité des méthodes `Affichage()` de la classe de base.



Si, par contre, le mot overloads est utilisé, nous pouvons avoir accès par la classe Salarie à toutes les méthodes surchargées ; y compris celles définies dans la classe Personne.



Substitution de méthode

Les classes dérivées héritent des propriétés et méthodes de leur classe de base. Vous pouvez donc les réutiliser, à partir d'une sous-classe, sans aucune modification. Par contre, si le fonctionnement de cette propriété ou méthode n'est pas adapté à la nouvelle classe, vous avez la possibilité de la substituer par une nouvelle implémentation dans la classe dérivée. Il faut dans ce cas utiliser le mot clé overrides lors de la substitution dans la classe dérivée. Il est également impératif que la classe de base ait autorisé cette substitution par l'utilisation du mot cléOverridable. Sans indication particulière, une méthode ou une propriété n'est pas substituable. En général, la substitution est utilisée pour assurer le polymorphisme entre classes. Les méthodes substituées doivent bien sûr avoir le même nom mais également le même nombre et type de paramètres que les méthodes de la classe de base auxquelles elles se substituent. Nous pouvons ainsi dans la classe Salarie substituer la méthode Affichage.

```
Public Overrides Sub affichage()
 Console.Write("Mr {0} {1} né le {2} gagne {3} euros par mois", nom, prenom,
date_naiss, salaire)
 End Sub
```

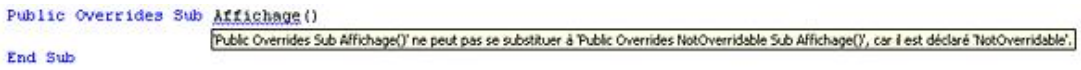
Avec cette déclaration, la méthode affichage de la classe Personne n'est plus visible aux utilisateurs de la classe Salarie. Seule la méthode Affichage de la classe Salarie est accessible. Cependant, le code de la méthode Affichage de la classe Salarie peut quand même avoir accès à cette méthode en utilisant le mot clé Mybase. Nous aurions donc pu écrire pour la méthode Affiche de la classe Salarie.

```
    Public Overrides Sub affichage()
        ' Appel de la méthode affichage de la classe Personne
        MyBase.affichage()
        ' Ajout des fonctionnalités spécifiques à la classe Salarie
        Console.WriteLine("gagne {0} euros par mois", salaire)
    End Sub
```

Dès qu'une méthode est déclarée comme étant substituable dans une classe, elle le sera pour toutes ses sous-classes quel que soit le degré de parenté (classe fille, petite fille...). Le mot cléNotOverridable peut être utilisé pour bloquer à partir d'un niveau donné cette fonctionnalité. Par exemple, dans la classe Salarie nous aurions pu écrire :

```
Public NotOverridable Overrides Sub affichage()
        MyBase.affichage()
        Console.WriteLine("gagne {0} euros par mois", salaire)
 End Sub
```

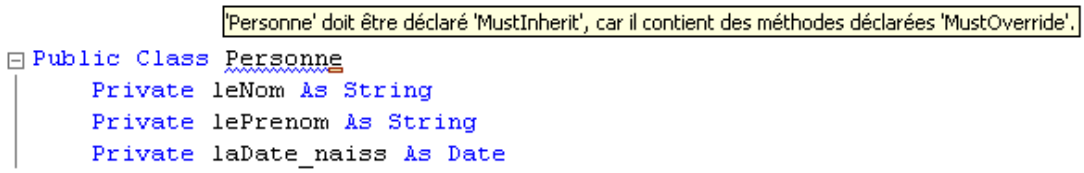
Cette syntaxe annule, pour les sous-classes de la classe Salarie, l'autorisation de substitution qui était mise en place par la classe Personne. Si nous essayons de substituer cette méthode dans une classe Chef qui hérite de Salarie nous obtenons le message suivant :



À l'inverse, nous pouvons exiger qu'une classe héritée substitue une méthode définie dans une classe de base. Cette méthode doit être marquée avec le mot cléMustOverride. Pour une telle méthode, il ne doit pas y avoir d'implémentation mais juste sa définition.

```
Public MustOverride Function etat_civil() As String
```

Une telle méthode est parfois appelée méthode abstraite. Elle exige que la classe dans laquelle elle est définie soit également marquée comme abstraite avec le mot clé MustInherit.



Masquage de méthode

Lorsque des éléments d'un programme partagent le même nom, l'un d'eux peut masquer l'autre. Dans un tel cas, l'élément masqué n'est plus accessible et le compilateur utilise à la place l'élément le masquant. Le masquage peut se faire entre des éléments de type différent. Seul le nom de l'élément est utilisé pour assurer le masquage. Lors du masquage, il convient d'utiliser le mot clé`Shadows`, devant le nom du membre assurant le masquage. Nous pouvons par exemple masquer la méthode `Affichage` dans une classe dérivée de la classe `Personne`.

```
Public Shadows Sub Affichage()

End Sub
```

Pour cette classe, il n'y a dorénavant qu'un seul élément appelé Affichage. Tout ce qui peut exister comme élément, dans la ou les classes de base de cette classe, et qui est nommé Affichage est masqué et inaccessible.

Cette technique est à utiliser avec précaution car en fonction de l'emplacement où se trouve une instruction, le même nom peut faire référence à des éléments de nature différente.

Méthode partielle

Les méthodes partielles sont utilisées pour nous permettre de personnaliser le code d'une classe partielle générée par un outil de Visual Studio. Elles sont principalement utilisées pour fournir une notification de changement. L'outil génère uniquement le squelette de la méthode et y fait appel lorsque la notification doit se produire. L'utilisateur de la classe peut éventuellement définir sa propre version de la méthode et dans ce cas celle-ci sera appelée à la place de celle générée automatiquement. Voyons comment mettre cela en application avec la classe `Personne`. Nous devons tout d'abord définir la classe comme étant une classe partielle puis prévoir à l'intérieur de la classe une méthode partielle en respectant les règles suivantes :

- la méthode doit être une procédure et non une fonction,
- le corps de la méthode doit être vide,
- la méthode doit être privée.

```
Partial Public Class Personne
    Private leNom As String
    Private lePrenom As String
    Private laDate_naiss As Date

    Partial Private Sub nomChanged()
    End Sub

    Public Property nom() As String
        Get
            Return leNom
        End Get
        Set(ByVal Value As String)
            leNom = Value
            nomChanged()
        End Set
    End Property
…
…
End Class
```

Il nous reste maintenant à personnaliser cette classe dans un autre fichier source et à tester le résultat. Pour cela, dans un module, ajoutons le code suivant :

```
Partial Class Personne
    Private Sub nomChanged()
        Console.WriteLine("un nouveau nom est affecté")
    End Sub
End Class

Module Module1
    Public Sub main()
        Dim p As Personne
        p = New Personne
        p.nom = "Dupont"
        Console.WriteLine(p.nom)
        Console.ReadLine()
    End Sub
End Module
```

À l'exécution, nous avons le résultat suivant :

```
un nouveau nom est affecté
Dupont
```

C'est bien le code de notre version de la méthode `nomChanged` qui vient d'être exécuter et pourtant nous n'avons pas touché le code original de la classe `Personne`.

Méthode d'extension

Les méthodes d'extension permettent l'ajout de fonctionnalités à une classe déjà définie sans bien sûr avoir à modifier le code de cette classe. Elles sont simplement écrites dans des modules et sont ensuite appelées exactement de la même façon que les méthodes disponibles directement dans la classe. Il y a cependant quelques règles à respecter :

- elles peuvent être de type procédure ou fonction mais pas de type propriété.
- Elles doivent être marquées avec l'attribut <Extension()>.
- Le type du premier paramètre de la méthode détermine le type étendu par cette méthode.
- Au moment de l'exécution ce premier paramètre représente l'instance de la classe sur laquelle la méthode est appelée.

Dans l'exemple ci-dessous, nous ajoutons une méthode à la classe `Personne`.

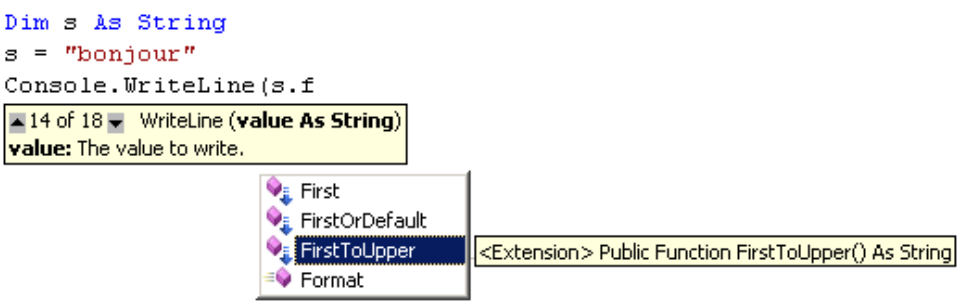
```
Module Module1
    <Extension()> Public Sub affichage(ByVal p As Personne)
        Console.WriteLine("nom : {0}", p.nom)
        Console.WriteLine("prénom : {0}", p.prenom)
        Console.WriteLine("date de naissance : {0}", p.date_naiss)
    End Sub

    Public Sub main()
        Dim p As Personne
        p = New Personne
        p.nom = "dupont"
        p.prenom = "paul"
        p.date_naiss = #6/7/1955#
        p.affichage()
        Console.ReadLine()
    End Sub
End Module
```

Les méthodes d'extension peuvent également être définies pour les types de base du Framework comme par exemple la classe String. Le code suivant ajoute à la classe String une méthode permettant de convertir le premier caractère d'une chaîne en majuscule.

```
<Extension()> Public Function FirstToUpper(ByVal s As String) As String
        If IsNothing(s) OrElse s.Length = 0 Then
            Return s
        ElseIf s.Length = 1 Then
```

```
            Return s.ToUpper
        Else
            Return s.Substring(0, 1).ToUpper & s.Substring(1, s.Length - 1)
        End If
    End Function
```

Si nous utilisons ensuite une variable de type String, notre nouvelle méthode devient disponible et est même proposée par IntelliSense.



À noter l'icône différente utilisée pour différencier une méthode d'extension d'une méthode normale de la classe.

e. Constructeurs et destructeurs

Les constructeurs sont des méthodes particulières d'une classe par différents aspects. Le constructeur s'appelle toujours `New` dans une classe. Il n'est jamais appelé explicitement dans le code mais de manière implicite, à la création d'une instance de classe. Comme pour une méthode classique, un constructeur peut attendre des paramètres. Le constructeur d'une classe qui n'attend pas de paramètre est désigné comme le constructeur par défaut de la classe. Le rôle du constructeur est principalement l'initialisation des champs d'une instance de classe. Les constructeurs peuvent également être surchargés mais, dans ce cas, l'utilisation du mot clé `overloads` est interdite.

Ajoutons à la classe `Personne` des constructeurs.

```
Public Sub New()
        leNom = ""
        lePrenom = ""
        leMotDePasse = ""
    End Sub
    Public Sub New(ByVal nom As String, ByVal prenom As String, ByVal pwd As String)
        leNom = nom
        lePrenom = prenom
        leMotDePasse = pwd
    End Sub
```

Lorsque nous créons une classe dérivée, elle peut aussi disposer de ses propres constructeurs. Si nous ajoutons dans la classe dérivée un constructeur par défaut, nous devons suivre quelques règles :

- Si dans la classe de base il existe un constructeur surchargé, nous devons appeler explicitement, dans le constructeur par défaut de la classe dérivée, le constructeur par défaut de la classe de base.
- S'il n'existe, dans la classe de base, que le constructeur par défaut, son appel est implicite dans le constructeur par défaut de la classe dérivée.
- Dans notre cas, le constructeur par défaut de la classe `Salarie` devrait avoir la forme suivante.

```
Public Sub New()
    MyBase.New()
    leSalaire = 0
End Sub
```

La ligne `MyBase.new()` provoque l'appel implicite du constructeur de la classe `Personne`. Cette ligne doit obligatoirement être la première du constructeur. Les destructeurs sont d'autres méthodes particulières d'une classe. Comme les constructeurs, ils sont appelés implicitement mais uniquement lors de la destruction d'une instance de classe. La signature du destructeur est imposée. Cette méthode doit être `protected`, se nommer `finalize` et ne prendre aucun paramètre. Du fait de cette signature imposée, il ne peut y avoir qu'un seul destructeur pour une classe, donc pas de surcharge possible pour les destructeurs.

La déclaration d'un destructeur est donc la suivante :

```
Protected Overrides Sub finalize()

End Sub
```

Le mot clé `overrides` est obligatoire car le destructeur se substitue au destructeur hérité de la classe `Object`. Le code présent dans le destructeur doit permettre la libération des ressources utilisées par la classe. On peut, par exemple, y trouver du code fermant un fichier ouvert par la classe ou la fermeture d'une connexion vers un serveur de base de données.

Nous verrons en détail dans le paragraphe "Destruction d'une instance", les circonstances dans lesquelles est appelé le destructeur.

---

f. Membres partagés

Les membres partagés sont des champs, propriétés ou méthodes qui sont accessibles par toutes les instances d'une classe. On parle également, dans certains langages, de membres statiques.

Ils sont très utiles lorsque vous avez à gérer, dans une classe, des informations qui ne sont pas spécifiques à une instance de classe mais à la classe elle-même. Par opposition aux membres d'instance, pour lesquels il existe un exemplaire par instance de la classe, les membres partagés existent en un exemplaire unique. La modification de la valeur d'un membre d'instance ne modifie la valeur que pour cette instance de classe alors que la modification de la valeur d'un membre partagé modifie la valeur pour toutes les instances de la classe. Les membres partagés sont assimilables à des variables globales, dans une application. Ils sont utilisables dans le code en y faisant référence par le nom de la classe ou grâce à une instance de la classe.

À noter que dans ce cas le compilateur ne trouve pas cette démarche logique et provoque un avertissement sur la ligne de code correspondante. Il est donc préférable de toujours utiliser le nom de la classe pour accéder à un membre partagé.

Les méthodes partagées suivent les mêmes règles et peuvent servir à la création de bibliothèques de fonctions. L'exemple classique est la classe `Math` dans laquelle de nombreuses fonctions partagées sont définies. Les méthodes partagées possèdent cependant une limitation car elles ne peuvent utiliser que des variables locales ou d'autres membres partagés de la classe. Elles ne doivent jamais utiliser des membres d'instance d'une classe, car il se peut que la méthode soit utilisée sans qu'il existe une instance de la classe. Le compilateur vérifiera ce genre d'erreurs.

Les membres partagés doivent être déclarés avec le mot clé `Shared`. Vous pouvez également, comme pour n'importe quel autre membre d'une classe, spécifier une visibilité. Par contre, une variable locale à une procédure ou fonction ne peut pas être partagée.

2. Utilisation d'une classe

L'utilisation d'une classe, dans une application, passe par deux étapes.

- La déclaration d'une variable permettant l'accès à l'objet.
- La création de l'objet.

Dans certains cas, ces deux étapes peuvent être omises, Visual Basic les effectuant lui- même. On parle alors d'objets intrinsèques. C'est le cas, par exemple, pour les objets de nom `My`.

a. Création d'une instance

Les variables objet sont des variables de type référence. Elles diffèrent des variables classiques par le fait que la variable ne contient pas directement les données mais une référence sur l'emplacement dans la mémoire de la machine où se trouvent les informations. Comme pour toutes les variables, elles doivent être déclarées avant leur utilisation. La déclaration s'effectue de manière identique à celle d'une variable classique (Integer ou autre).

```
Dim p as Personne
```

Après cette étape, la variable existe mais ne référence pas d'emplacement valide. Elle contient la valeur `Nothing`.

La deuxième étape consiste réellement à créer l'instance de la classe. Le mot clé `New` est utilisé à cet effet. Il attend comme paramètre le nom de la classe dont il est chargé de créer une instance. L'opérateur `New` fait une demande au système pour obtenir la mémoire nécessaire au stockage de l'instance de la classe, puis initialise la variable avec cette adresse mémoire. Le constructeur de la classe est ensuite appelé pour initialiser la nouvelle instance créée.

```
p = New Personne
```

Les deux opérations peuvent être combinées en une seule ligne.

```
Dim p as New Personne
```

Le constructeur par défaut est appelé, dans ce cas. Pour utiliser un autre constructeur, vous devez spécifier une liste de paramètres et en fonction du nombre et du type des paramètres, l'opérateur `New` appelle le constructeur correspondant.

```
Dim p as New Personne("Dupond","Paul",#12/24/1953#)
```

b. Initialisation d'une instance

Après avoir créé une instance de classe, vous pouvez initialiser les membres de celle-ci par l'intermédiaire des propriétés de classe. Il est possible de combiner ces deux étapes en une seule. Pour cela, lors de la création de l'instance, il faut fournir une liste de propriétés et de valeurs à affecter à ces propriétés. Voici ci-dessous la syntaxe exacte à utiliser :

```
Dim p As Personne
p = New Personne With {.nom = "dupont", .prenom = "paul"}
```

Il n'y a pas de limitation sur le nombre de propriétés initialisées ni sur l'ordre d'apparition des propriétés dans la liste d'initialisation. Cette unique ligne de code est l'équivalent de la syntaxe moins condensée et plus traditionnelle suivante :

```
Dim p As Personne
   p = New Personne
   With p
       .nom = "Dupont"
       .prenom = "Paul"
   End With
```

c. Destruction d'une instance

La destruction d'une instance de classe est automatique dans une application. Le Common Language Runtime surveille à intervalles réguliers que toutes les instances de classes, créées dans l'application, sont encore accessibles ; c'est-à-dire qu'il existe encore dans l'application une variable ou une propriété permettant l'accès à cette instance. Si aucun moyen d'accéder à cette instance n'est trouvé, l'objet est alors marqué comme étant orphelin. Lorsque les réserves mémoire de l'application deviennent trop faibles, le Garbage Collector intervient et élimine les objets orphelins. C'est lors de cette élimination que les destructeurs de chacun des objets sont appelés. Il n'y a aucun moyen de précipiter les choses en demandant l'élimination immédiate de la mémoire d'une instance particulière de classe. Cette situation est parfois problématique lorsque l'objet utilise une ressource externe comme, par exemple, une connexion vers un serveur de base de données. Si la fermeture de la connexion est prévue dans le destructeur de la classe, il peut se passer assez longtemps entre le moment où l'objet devient inaccessible et celui où le destructeur est appelé.

Pour pallier ce problème, il est possible de mettre en œuvre une autre solution. Le code chargé de la libération des ressources est placé dans une autre méthode et cette méthode est appelée explicitement dans le code. En général, cette méthode est nommée `Dispose`. Pour être certain que les ressources sont bien libérées, vous pouvez également prévoir un appel à cette méthode dans le destructeur de la classe.

Un autre problème peut alors survenir : si la méthode a été appelée explicitement dans le code de l'application, elle le sera à nouveau, de manière implicite, lorsque le Garbage Collector entrera en action. Vous devez, donc, faire en sorte que le code de cette méthode `Dispose` puisse être exécuté deux fois sans causer d'erreur. Vous pouvez également indiquer au Garbage Collector, qu'il ne doit pas exécuter le destructeur de cette instance de classe. Pour cela, dans la méthode `Dispose`, vous devez le prévenir que le travail de "nettoyage" est déjà réalisé, en appelant la méthode`SuppressFinalize`. Le code de la méthode `Dispose` et du destructeur doit alors avoir la forme suivante :

```
Public Sub Dispose()
        ' inserer le code chargé de la libération des ressources
```

```
        '
        ' Demande au Garbage Collector de ne pas appeler le destructeur
        GC.SuppressFinalize(Me)
    End Sub
    Protected Overrides Sub finalize()
        Dispose()
    End Sub
```

Une dernière solution consiste à demander au Garbage Collector d'entrer en action immédiatement en appelant la méthode `collect`.

```
GC.Collect()
```

Cette solution n'est cependant pas recommandée, car elle consomme énormément de ressources sur le système pour parfois ne récupérer que quelques centaines d'octets en mémoire.

d. Liaison tardive, liaison précoce

Le compilateur Visual Basic effectue une opération appelée liaison lorsqu'un objet est affecté à une variable. Cette liaison est dite précoce lorsque la variable est créée à partir d'une classe spécifique. Cette fonctionnalité permet au compilateur d'effectuer des optimisations sur le code généré. L'affection d'un objet peut aussi se réaliser à une variable de type Object. Ce type de variable est capable de référencer n'importe quel autre type de classe. Dans ce cas, la liaison est dite tardive car le type réel de l'objet ne sera découvert qu'à l'exécution de l'application. Cette technique est à éviter car elle génère un code moins efficace et surtout elle ne permet pas de bénéficier de la complétion automatique du code dans l'éditeur ni de l'aide dynamique. En effet, dans ce cas Visual Basic ne peut pas déterminer le type réel de l'objet manipulé.

Cependant, certaines fonctions retournent un type Object, mais pour pouvoir le manipuler, il convient de prendre quelques précautions. La première solution consiste à n'utiliser, avec la valeur renvoyée par la fonction, que des membres de la classe Object. Cette solution est relativement limitative quant aux fonctionnalités disponibles.

La deuxième solution consiste à affecter, à une variable d'un type particulier, la valeur renvoyée par la fonction. Cette solution permet d'utiliser toutes les fonctionnalités de l'objet retourné par la fonction. Cependant, il faut être certain que l'objet retourné est bien une instance de la classe que l'on souhaite manipuler. D'ailleurs, le compilateur se chargera de nous le rappeler.

```
Dim fils As Personne
        Option Strict On interdit les conversions implicites de 'Object' en 'ConsoleApplication1.Personne'.
fils = p.getEnfant(0)
```

Nous devons donc nous assurer du type de l'objet retourné et demander explicitement le transtypage. Nous pouvons obtenir le nom du type de l'objet et effectuer une comparaison de chaîne de caractères.

```
    Dim fils As Personne
     If TypeName(p.getEnfant(0)).Equals("Personne") Then
          fils = CType(p.getEnfant(0), Personne)
     End If
```

Cette solution fonctionne mais comporte le risque de mal orthographier le nom de la classe lors de la comparaison. L'opérateur`TypeOf ... Is ...` est plus adapté à cette situation.

```
    'Dim fils As Personne
     If TypeOf p.getEnfant(0) Is Personne Then
            fils = CType(p.getEnfant(0), Personne)
     End If
```

À noter que le transtypage ne change pas le type de l'objet en mémoire mais permet simplement de le voir d'une autre façon. Si par exemple nous avons en mémoire une instance de la classe `Salarie`, le transtypage nous permet de le voir comme un Object, une Personne ou un Salarie mais cela restera toujours une instance de la classe `Salarie`.

3. Héritage

L'héritage est une puissante fonctionnalité d'un langage orienté objet mais peut parfois être utilisée mal à propos. Deux types de relations peuvent être utilisés entre deux classes. Nous pouvons avoir la relation "est une sorte de" et la relation "concerne un". La relation d'héritage doit être envisagée lorsque la relation "est une sorte de" peut être appliquée entre deux classes. Prenons un exemple avec trois classes : Personne, Client, Commande.

Essayons les relations pour chacune des classes.

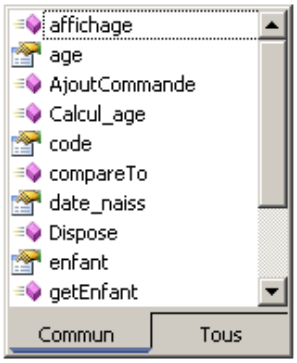
1. Une commande est une sorte de client
2. Une commande est une sorte de personne
3. Un client est une sorte de commande
4. Un client est une sorte de personne
5. Une personne est une sorte de client
6. Une personne est une sorte de commande

Parmi toutes ces tentatives, il n'y en a qu'une seule qui nous semble logique : un client est une sorte de personne. Nous pouvons donc envisager une relation d'héritage entre ces deux classes. La mise en œuvre est très simple au niveau du code puisque, dans la déclaration de la classe, il suffit juste de spécifier le mot clé `Inherits` suivi du nom de la classe dont on souhaite hériter. Visual Basic n'acceptant pas l'héritage multiple, vous ne pouvez spécifier qu'un seul nom de classe de base.

```
Public Class Client
    Inherits Personne
    Private leCode As Integer
    Public Property code() As Integer
        Get
            Return leCode
        End Get
        Set(ByVal value As Integer)
            leCode = value
        End Set
    End Property
End Class
```

La classe peut ensuite être utilisée et propose toutes les fonctionalités définies dans la classe `Client` plus celles héritées de la classe `Personne`.

```
Dim c As Client
c = New Client("dupond", "paul", "secret", 12345)
c.
```



a. MyBase et MyClass

Il est légitime de vouloir ensuite modifier le fonctionnement de certaines méthodes héritées pour les adapter à la classe `Client`. Par exemple, la méthode `Affichage` peut être substituée pour tenir compte des nouveaux champs disponibles dans la classe.

```
    Public Overrides Sub affichage()
        Console.Write("Mr {0} {1} né le {2}", nom, prenom, date_naiss)
        Console.Write(" code Client : {0}", leCode)
    End Sub
```

Ce code fonctionne très bien mais ne respecte pas l'un des principes de la programmation objet qui veut que l'on réutilise au maximum ce qui existe déjà. Dans notre cas, nous avons déjà une portion de code chargée de l'affichage du nom, du prénom et de la date de naissance d'une personne. Pourquoi ne pas la réutiliser dans la méthode `affichage` de la classe `Client` puisque l'on en hérite ?

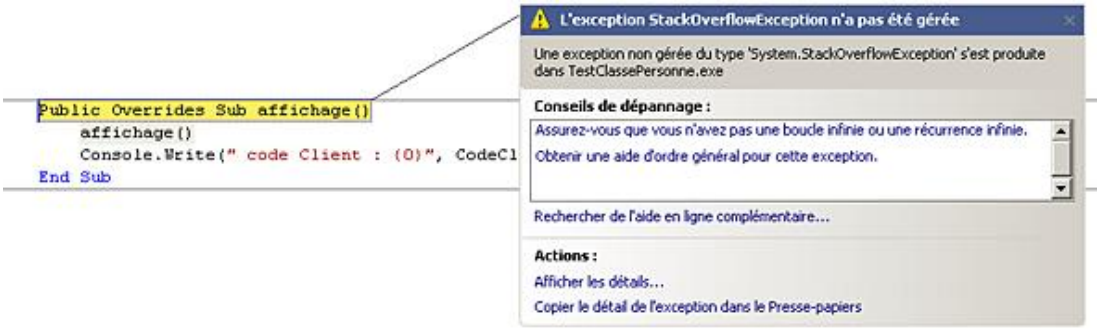
Notre méthode devient donc :

```
    Public Overrides Sub affichage()
        affichage()
        Console.Write(" code Client : {0}", leCode)
    End Sub
```

Essayons de l'utiliser :

```
     Dim c As Client
        c = New Client
        c.nom = "dupond"
        c.prenom = "paul"
        c.date_naiss = #12/23/1954#
        c.code = 12345
        c.affichage()
```

Hélas, le résultat n'est pas à la hauteur de nos espérances !



Que s'est-il passé lors de l'exécution ?

Lors de l'appel de la méthode `affichage`, la première ligne de code a consisté à appeler la méthode `affichage` de la classe de base. En fait, Visual Basic recherche la première méthode `affichage` qu'il trouve et appelle ainsi en boucle la méthode `affichage` de la classe `Client` d'où l'erreur de débordement de pile que nous obtenons.

Pour éviter ce genre de problème, nous devons lui préciser que la méthode `affichage` à appeler se trouve dans la classe de base. Pour cela, nous devons utiliser le mot clé `MyBase` pour qualifier la méthode `affichage` appelée.

```
    Public Overrides Sub affichage()
        MyBase.affichage()
        Console.Write(“ code Client : {0}”, leCode)
    End Sub
```

Après cette modification, tout rentre dans l'ordre et notre code affiche.

```
Mr DUPOND paul né le 23/12/1954 00:00:00 code Client : 12345
```

La même syntaxe peut être utilisée pour appeler le constructeur de la classe de base.

L'appel au constructeur de la classe de base doit être la première ligne d'un constructeur.

Nous pouvons donc créer un constructeur pour la classe `Client` qui utilise le constructeur de la classe `Personne`.

```
Public Sub New(ByVal nom As String, ByVal prenom As String,
ByVal pwd As String, ByVal leCode As Integer)
        MyBase.New(nom, prenom, pwd)
       leCode = leCode
    End Sub
```

Vérifions que le nouveau constructeur fonctionne :

```
        Dim c As Client
        c = New Client("dupond", "paul", "secret", 12345)
        c.date_naiss = #12/23/1967#
        c.affichage()
        Console.ReadLine()
```

Nous affiche :

```
Mr dupond paul né le 23/12/1967 00:00:00 code Client : 0
```

Les informations ont bien été prises en compte, sauf le code client qui reste à zéro. Regardons de plus près le code du constructeur. Nous découvrons qu'un paramètre du constructeur porte le même nom qu'un champ de la classe. Lorsque nous écrivons la ligne `CodeCli=CodeCli` le compilateur considère que nous souhaitons affecter au paramètre `CodeCli` la valeur contenue dans le paramètre `CodeCli`. Rien d'illégal, mais ce n'est absolument pas ce que nous souhaitons faire. Nous devons indiquer que l'affectation doit se faire à la variable membre de la classe. Pour cela, nous devons la préfixer avec le mot clé`MyClass`.

Le constructeur devient donc :

```
    Public Sub New(ByVal nom As String, ByVal prenom As String, ByVal pwd As
String, ByVal leCode As Integer)
        MyBase.New(nom, prenom, pwd)
        MyClass.leCode = leCode
    End Sub
```

Notre code de test nous affiche alors les bonnes informations :

```
Mr dupond paul né le 23/12/1967 00:00:00 code Client : 12345
```

b. Classes abstraites

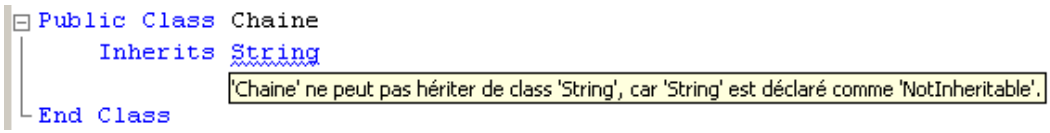
Les classes abstraites sont des classes qui peuvent uniquement être utilisées comme classe de base dans une relation d'héritage. Il est impossible de créer une instance d'une classe abstraite. Elles servent essentiellement de modèle pour la création de classe, devant toutes avoir un minimum de caractéristiques identiques. Elles peuvent contenir des champs, des propriétés et des méthodes comme une classe ordinaire. Cette technique facilite l'évolution de l'application, car si une nouvelle fonctionnalité doit être disponible dans les classes dérivées, il suffit d'ajouter cette fonctionnalité dans la classe de base. Il est également possible de ne pas fournir d'implémentation pour une classe abstraite et ainsi laisser à l'utilisateur de la classe le soin de créer l'implémentation dans la classe dérivée.

Pour qu'une classe devienne une classe abstraite, vous devez utiliser le mot clé`MustInherit` dans la déclaration de la classe.

```
Public MustInherit Class Modele

End Class
```

c. Classes finales

Les classes finales sont des classes ordinaires qui peuvent être instanciées mais ne sont pas utilisables comme classe de base dans une relation d'héritage. C'est le cas de plusieurs classes du Framework.NET, comme par exemple la classe `String`.



Une classe finale doit être définie avec le mot clé`NotInheritable`.

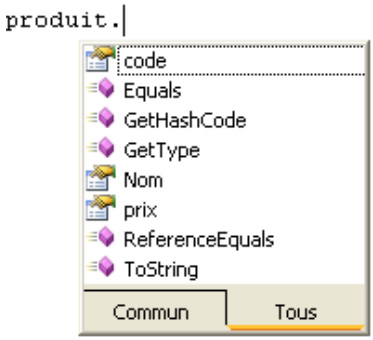
```
Public NotInheritable Class Client
    Inherits Personne
End Class
```

d. Classes anonymes

Visual Basic nous fournit la possibilité de créer des objets sans écrire de code pour la définition de la classe correspondante. C'est le compilateur qui va prendre en charge la génération de la classe. Évidemment il va falloir l'aider un petit peu et lui fournir quelques informations pour qu'il puisse générer la classe et en créer une instance. La déclaration d'une instance de classe anonyme se base principalement sur le principe vu au paragraphe b concernant l'initialisation d'une instance. Lors de la déclaration d'une telle instance nous devons fournir la liste des propriétés souhaitées ainsi que leurs valeurs. Voyons donc un premier exemple.

```
Dim produit = New With {.Nom = "biscuit", .Prix = 1.56}
```

La variable produit fait référence à une instance de classe contenant deux propriétés Nom et Prix. Ces deux propriétés sont initialisées respectivement avec les valeurs "biscuit" et 1.56. Pour vérifier cela, demandons encore une fois à IntelliSense son avis.



Il nous indique bien que pour notre variable produit, les deux propriétés sont disponibles. Par contre il nous propose également plusieurs méthodes. D'où viennent-elles ? En fait, les classes anonymes respectent les mêmes règles que toutes les autres classes car elles héritent par défaut de la classe `Object`. Les méthodes proposées sont donc celles héritées de la classe `Object`. Ce sont d'ailleurs les seules qu'il pourra y avoir dans une classe anonyme car pour ce genre de classe il est impossible de définir autre chose que des propriétés, donc pas de méthodes ni d'événements. Vous devez certainement vous poser la question : Comment le compilateur fait-il pour s'y retrouver s'il existe plusieurs instances de classe anonyme ? En fait, ces classes sont anonymes pour nous mais pas pour lui car pendant la compilation il va générer automatiquement un nom pour ces classes. Pour le vérifier, demandons à la variable produit de nous indiquer son type.

```
Console.WriteLine(produit.GetType.Name)
```

Nous obtenons la réponse suivante à l'exécution.

```
VB$AnonymousType_0`2
```

Il existe donc bien un nom pour ces classes. Par contre, ce nom n'est connu qu'après la compilation donc hors de question de l'utiliser dans notre code. Ce serait d'ailleurs une très mauvaise idée car il est susceptible de changer à chaque compilation de l'application.

En fait, le compilateur ne crée pas une nouvelle classe pour chaque instanciation car il vérifie, avant de créer une nouvelle classe, s'il n'a pas déjà traité une définition identique. Pour cela, il se base sur les éléments suivants :

- le nombre de propriétés ;
- le nom des propriétés ;
- le type des propriétés ;
- l'ordre des propriétés dans la définition.

Si tous ces éléments sont identiques alors il réutilisera la classe qu'il a déjà générée pour créer l'instance. Vérifions cela :

```
Dim produit1 = New With {.Nom = "biscuit", .prix = 1.56}
Dim produit2 = New With {.Nom = "confiture", .prix = 2.24}
Console.WriteLine("la classe de produit1 est :{0}", produit1.GetType.Name)
Console.WriteLine("la classe de produit2 est :{0}", produit2.GetType.Name)
```

Nous obtenons le résultat suivant :

```
la classe de produit1 est :VB$AnonymousType_0`2
la classe de produit2 est :VB$AnonymousType_0`2
```

Tout est normal car les deux variables ont rigoureusement la même définition. Si nous faisons une petite modification en inversant simplement l'ordre des propriétés :

```
Dim produit1 = New With {.Nom = "biscuit", .prix = 1.56}
```

```
Dim produit2 = New With {.prix = 2.24, .Nom = "confiture"}
Console.WriteLine("la classe de produit1 est :{0}", produit1.GetType.Name)
Console.WriteLine("la classe de produit2 est :{0}", produit2.GetType.Name)
```

Nous obtenons cette fois le résultat suivant :

```
la classe de produit1 est :VB$AnonymousType_0`2
la classe de produit2 est :VB$AnonymousType_1`2
```

Le compilateur a cette fois généré deux classes puisque les définitions sont différentes.

Lors de la déclaration d'une instance de classe anonyme, certaines propriétés peuvent être définies comme des propriétés clés de l'instance. Ces propriétés ont un rôle important pour la comparaison d'instances de classes anonymes. Elles sont identifiées grâce au mot clé 'key' placé devant le nom de la propriété. Les propriétés déclarées ainsi sont en lecture seule.

```
Dim produit = New With {Key .code = 12854, .Nom = "biscuit", .prix = 1.56}

produit.code = 98654
Property 'code' is 'ReadOnly'.
```

Pour comparer deux instances de classe anonyme, nous devons utiliser la méthode `Equals`. Cette méthode est héritée de la classe `Object` mais substituée par le compilateur au moment de la génération de la classe anonyme. Cette méthode indiquera que deux instances sont identiques si toutes les conditions suivantes sont respectées.

- Les deux objets doivent être déclarés dans le même assembly.
- Les propriétés doivent avoir le même nom, le même type et être déclarées dans le même ordre.
- Il doit y avoir au moins une propriété déclarée avec le mot clé `Key`.
- Les mêmes propriétés doivent être déclarées avec le mot clé `Key` dans les deux instances.
- Les valeurs des propriétés déclarées avec le mot clé `Key` doivent être identiques dans les deux instances.
- Une instance de classe anonyme qui n'a pas de propriété déclarée avec le mot clé `Key` ne peut être égale qu'à elle-même.

Vérifions ces règles en comparant différents produits.

```
Dim produit1 = New With {Key .code = 12854, .Nom = "biscuit", .prix = 1.56}
Dim produit2 = New With {Key .code = 12854, .Nom = "biscuit", .prix = 1.56}

Console.WriteLine(produit1.Equals(produit2))
```

Ce code nous affiche `True` car les deux instances respectent l'ensemble des règles d'égalité citées ci-dessus.

Essayons d'enfreindre une des règles en oubliant par exemple le mot clé `Key`.

```
Dim produit1 = New With {.code = 12854, .Nom = "biscuit", .prix = 1.56}
Dim produit2 = New With {Key .code = 12854, .Nom = "biscuit", .prix = 1.56}

Console.WriteLine(produit1.Equals(produit2))
```

Le résultat est sans appel : les deux instances ne sont plus identiques.

4. Interfaces

Nous avons vu que l'on pouvait obliger une classe à implémenter une méthode en la déclarant avec le mot clé `MustOverride`. Si nous avons plusieurs classes qui doivent implémenter la même méthode, il est plus pratique d'utiliser les interfaces. Comme les classes, les interfaces permettent de définir un ensemble de propriétés, méthodes, événements. Cependant, elles ne contiennent aucun code. L'implémentation doit s'effectuer au niveau de la classe elle-même. L'interface constitue un contrat que vous signez. En déclarant que votre classe implémente une interface, vous vous engagez à fournir, dans votre classe, tout ce qui est défini dans l'interface. Il convient d'être prudent si vous

utilisez les interfaces et de ne jamais modifier une interface déjà utilisée sinon vous courez le risque de devoir reprendre le code de toutes les classes qui implémentent cette interface.

Pour pouvoir utiliser une interface, il convient de la définir au préalable. La déclaration est similaire à la déclaration d'une classe mais en utilisant les mots clés `Interface` et `End Interface`.

Vous pouvez éventuellement utiliser le mot clé `Inherits` pour introduire une relation d'héritage dans votre interface. Les seules instructions qui doivent apparaître dans une interface sont des déclarations de propriétés avec `Property`, de procédures et fonctions avec `Sub` et `Function`, ou d'événements avec `Event`. Il ne doit y avoir aucun code dans les procédures et fonctions même pas de `End Sub` ni de `End Function`. Créons donc notre première interface.

```
    Public Interface Comparable
        Function compare(ByVal o1 As Object) As Integer
    End Interface
```

Cette interface nous obligera à créer dans les classes qui l'implémenteront une fonction nous permettant de comparer l'instance courante d'un objet et l'objet qui sera passé comme paramètre. La fonction retournera une valeur égale à 1 si l'objet passé comme paramètre est supérieur à l'instance courante, une valeur égale à zéro si les deux objets sont égaux, une valeur égale à -1 si l'instance courante est supérieure à l'objet passé comme paramètre.

Mais quels critères allons-nous utiliser pour dire qu'un objet est supérieur à un autre ?

Dans la description de notre interface, ce n'est pas notre souci ! Nous laissons le soin à la personne qui va définir une classe utilisant notre l'interface de définir quels sont les critères de comparaison. Par exemple, dans notre classe `Client`, nous pourrions implémenter l'interface comparable de la manière suivante en choisissant de comparer deux clients sur le nom :

```
Public Class Client
    Inherits Personne
    Implements Comparable

...
...

Public Function compare(ByVal o1 As Object) As Integer
Implements Comparable.compare
        Select Case nom
            Case Is < o1.nom
                Return -1
            Case o1.nom
                Return 0
            Case Else
                Return 1
        End Select
End Function
```

Deux modifications sont visibles dans la classe :

- le fait qu'elle implémente l'interface `Comparable`.
- L'implémentation réelle de la fonction `compare` avec l'indication de l'interface et de la méthode qu'elle implémente.

Dans cette fonction, la comparaison se fera sur le nom des clients. Très bien mais ça sert à quoi ?

Il arrive fréquemment que l'on ait besoin de trier des éléments dans une application. Deux solutions :

- créer une fonction de tri spécifique pour chaque type d'élément que l'on veut trier.
- Créer une routine de tri générique et faire en sorte que les éléments que l'on utilise soient triables par cette routine.

Les interfaces vont nous aider à mettre en œuvre cette deuxième solution. Pour pouvoir trier des éléments, et quelle que soit la méthode utilisée pour le tri, nous aurons besoin de comparer deux éléments. Pour être certain que notre routine de tri fonctionnera sans problème, il faut s'assurer que les éléments qu'elle devra trier auront la possibilité d'être comparés les uns aux autres. Nous ne pouvons garantir cela que si tous nos éléments implémentent l'interface `Comparable`. Nous allons donc l'exiger dans la déclaration de notre routine de tri.

```
Public Sub tri(ByVal tablo As Comparable())
```

Définie ainsi, notre procédure sera capable de trier toutes sortes de tableaux pourvu que leurs éléments implémentent l'interface `Comparable`. Nous pouvons donc écrire le code suivant et utiliser la méthode `compare` sans risque.

```
    Public Sub tri(ByVal tablo As Comparable())
        Dim i As Integer
        Dim j As Integer
        Dim o As Object
        For i = 0 To UBound(tablo) - 1
            For j = i + 1 To UBound(tablo)
                If (tablo(j).compare(tablo(i))) < 0 Then
                    o = tablo(j)
                    tablo(j) = tablo(i)
                    tablo(i) = o
                End If
            Next
        Next
    End Sub
```

Puis pour tester notre procédure, créons quelques clients, essayons de les trier et puis d'afficher leurs noms.

```
        Dim tab(4) As Client
        tab(0) = New Client("toto2", "prenom2 ", "secret", 2)
        tab(1) = New Client("toto1", "prenom1 ", "secret", 1)
        tab(2) = New Client("toto5", "prenom5 ", "secret", 5)
        tab(3) = New Client("toto3", "prenom3 ", "secret", 3)
        tab(4) = New Client("toto4", "prenom4 ", "secret", 4)
        tri(tab)
        Dim i As Integer
        For i = 0 To 4
            Console.WriteLine(tab(i).nom)
        Next
```

Nous obtenons le résultat suivant :

```
Mr toto1 prenom1 né le 01/01/0001 00:00:00 code Client : 1
Mr toto2 prenom2 né le 01/01/0001 00:00:00 code Client : 2
Mr toto3 prenom3 né le 01/01/0001 00:00:00 code Client : 3
Mr toto4 prenom4 né le 01/01/0001 00:00:00 code Client : 4
Mr toto5 prenom5 né le 01/01/0001 00:00:00 code Client : 5
```

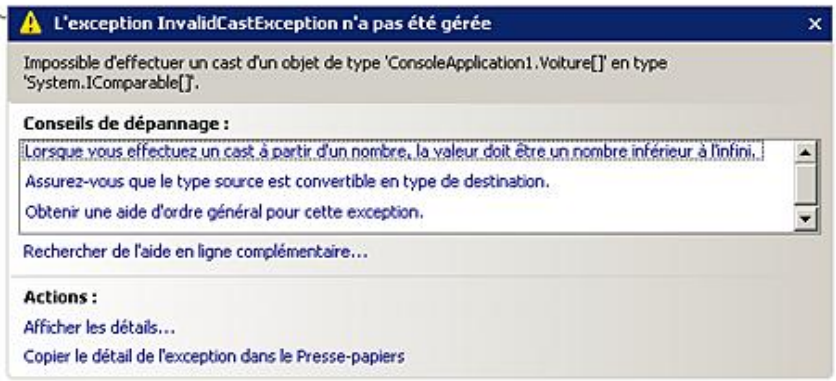
Nous avons bien la liste de nos clients triée par ordre alphabétique sur le nom.

Essayons d'utiliser notre procédure de tri avec un tableau d'objets qui n'implémentent pas l'interface `Comparable`.

```
Dim tab(4) As Voiture
tab(0) = New Voiture("Fiat", "500")
tab(1) = New Voiture("Peugeot","205")
tab(2) = New Voiture("Opel", "Corsa")
tab(3) = New Voiture("Renault", "Laguna")
tab(4) = New Voiture(Lancia", "Delta")
tri(tab)
Dim i As Integer
For i = 0 To 4
    Console.WriteLine(tab(i).nom)
Next
```

Pas de problème à la compilation, par contre les choses se compliquent pendant l'exécution.

```
Dim tabVoiture(4) As Voiture
tabVoiture(0) = New Voiture("Fiat", "500")
tabVoiture(1) = New Voiture("Peugeot", "205")
tabVoiture(2) = New Voiture("Opel", "Corsa")
tabVoiture(3) = New Voiture("Renault", "Laguna")
tabVoiture(4) = New Voiture("Lancia", "Delta")
tri(tabVoiture)
```



Cette erreur intervient au moment de l'appel de la procédure de tri. Les éléments du tableau que nous avons passé comme paramètre n'implémentent pas l'interface `Comparable` et nous ne sommes pas certains qu'ils contiennent une fonction `compare`. À noter que, même s'il existe une fonction `compare` correcte dans la classe `voiture`, il faut obligatoirement spécifier que cette classe implémente l'interface `Comparable`, pour que notre code puisse fonctionner.

5. Les événements

Les méthodes nous permettent de communiquer avec les objets qui composent une application mais les objets ont également la possibilité de nous faire part de leurs réactions en générant des événements. Ces événements doivent ensuite être pris en compte pour réagir à ce qui vient de se passer dans l'application.

Les événements sont très largement utilisés dans la conception de l'interface graphique d'une application car ils nous permettent d'avoir des informations sur les actions effectués par l'utilisateur de l'application.

a. Déclaration et déclenchement d'événements

Voyons, tout d'abord, comment générer un événement dans une classe. La première chose à faire est de déclarer l'événement dans la classe. Cette déclaration s'effectue de la même manière que celle des variables internes à la classe. On utilise pour cela le mot clé `event` suivi du nom de l'événement. Nous pouvons également, lors de la déclaration, spécifier des paramètres qui fourniront des informations supplémentaires sur l'événement lui-même ou sur l'état de l'objet, au moment où l'événement se déclenche. Nous pouvons, par exemple créer dans notre classe `Client` un événement qui se déclenchera à chaque ajout d'une nouvelle commande. Comme information supplémentaire, notre événement fournira le nombre de commandes après l'ajout.

```
Public Event NouvelleCommande(ByVal nbCommandes As integer)
```

Il nous reste maintenant à définir dans quelles conditions notre événement sera déclenché et bien sûr à le déclencher à ce moment-là. Pour notre classe `Client`, le plus évident est de générer notre événement au moment où l'on exécute la méthode permettant d'ajouter une commande. Pour cela, nous allons créer la méthode `AjoutCommande`. Cette méthode reçoit comme paramètre le numéro de la commande à ajouter. La seule action de cette méthode sera d'incrémenter le nombre de commandes du client.

```
Public Sub AjoutCommande(ByVal numero As Integer)
        nbCommandes = nbCommandes + 1
End Sub
```

Dans la méthode `AjoutCommande`, nous allons donc déclencher l'événement avec l'instruction`RaiseEvent` suivie du nom de l'événement déclaré au préalable dans la classe. Nous devons également fournir les éventuels paramètres déclarés avec notre événement.

```
Public Sub AjoutCommande(ByVal numero As Integer)
        nbCommandes = nbCommandes + 1
        RaiseEvent NouvelleCommande(nbCommandes)
End Sub
```

Après l'exécution de l'instruction `RaiseEvent`, l'événement sera transmis à tous les éléments qui se seront déclarés

intéressés par cet événement. Si plusieurs éléments sont intéressés par l'événement, ils recevront les uns après les autres la notification de l'événement.

À noter toutefois qu'il n'est pas possible de prédire l'ordre dans lequel les éléments de notre application seront prévenus, ni de prévenir un nouvel élément tant que le précédent n'aura pas fini de traiter l'événement.

b. Gérer les événements

Regardons maintenant comment récupérer les événements dans notre application. Deux solutions sont possibles :

- déclarer une variable avec le mot clé `WithEvents` et utiliser la clause `Handles`.
- Ajouter manuellement un gestionnaire d'événement.

La deuxième solution est plus souple car elle nous permettra d'ajouter ou de supprimer dynamiquement des gestionnaires d'événement pendant le fonctionnement de notre application, mais elle nécessite un peu plus de code que la première solution.

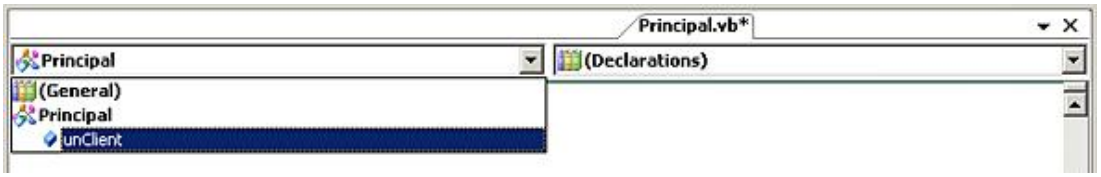
Utilisation de la clause Handles

Cette solution nécessite une déclaration de variable spécifique. La déclaration d'une variable pouvant référencer une instance de la classe `Client` s'effectue de la manière suivante.

```
Dim WithEvents unClient As Client
```

Il convient maintenant de décider quel code sera exécuté lorsque cet événement se produira. Une procédure liée à la réception de cet événement doit être créée. Pour cela, l'environnement de développement nous facilite la tâche. En haut de notre fenêtre de code, nous avons deux listes disponibles.

La première nous indique toutes les sources d'événements disponibles et nous permet donc de choisir d'où proviendra notre événement.



La seconde indique, pour la source d'événement actuellement sélectionnée, quels sont les événements susceptibles d'être déclenchés.

Après avoir fait notre choix dans les deux listes, l'environnement génère automatiquement le squelette de la procédure qui sera appelée par le déclenchement de l'événement.

```
Private Sub unClient_NouvelleCommande(ByVal nbCommandes As Integer) Handles
unClient.NouvelleCommande
End Sub
```

Cette méthode est identique à celles que l'on a écrit jusqu'à présent, avec en plus le mot clé `Handles` suivi du nom de l'événement qui sera traité par cette procédure. Par convention, le nom de la procédure est constitué par le nom de l'élément suivi par le nom de l'événement géré sur cet objet. Mais ceci reste une convention permettant de faciliter la lecture du code. La seule obligation est que la procédure respecte la signature définie dans l'événement.

Nous pouvons également ajouter plusieurs procédures qui seront appelées par le même événement, mais il faut dans ce cas reprendre le code manuellement et bien sûr utiliser des noms différents pour les procédures. Cette méthode manque de souplesse car il est impossible de choisir, pendant l'exécution, quel gestionnaire d'événement sera utilisé.

Il existe également une autre contrainte liée au fait que les variables déclarées `WithEvents` doivent obligatoirement être déclarées en dehors de toute procédure ou fonction.

Ajout et suppression de gestionnaire d'événement

Pour éviter ces limitations, nous avons la possibilité d'ajouter manuellement des gestionnaires d'événement. Dans ce cas, la variable doit être déclarée sans le mot clé `WithEvents`. On utilise ensuite le mot clé`AddHandler` pour associer à un événement une procédure pour le gérer.

```
AddHandler unClient.NouvelleCommande, AddressOf unClient_NouvelleCommande
```

Le premier paramètre de `AddHandler` est l'événement que l'on veut gérer puis on lui indique également, par l'intermédiaire de `AddressOf`, l'emplacement où se trouve la procédure qui sera exécutée pour gérer cet événement. Nous pouvons ainsi décider, au cours du fonctionnement de l'application, qui gèrera un événement particulier. Par exemple, si nous avons une application qui fonctionne en mode console ou en mode Windows, nous aurons un mode d'affichage différent.

```
        Dim mode As String = "console"
        Dim unClient As Client
        unClient = New Client("Dupond", "paul ", "secret", 12345)

        If mode = "graphique" Then
            AddHandler unClient.NouvelleCommande, AddressOf gestion_graph
        Else
            AddHandler unClient.NouvelleCommande, AddressOf gestion_texte
        End If
        unClient.AjoutCommande(10)

    Public Sub gestion_graph(ByVal nbcmd As Integer)
        MsgBox("nombre de commandes  " & nbcmd)
    End Sub

    Public Sub gestion_texte(ByVal nbcmd As Integer)
        Console.WriteLine(“nombre de commandes " & nbcmd)
    End Sub
```

De la même manière, nous pouvons supprimer un gestionnaire d'événement qui est déjà référencé, en utilisant`RemoveHandler`.

```
RemoveHandler unClient.NouvelleCommande, AddressOf gestion_graph
```

Cette instruction ne supprime pas la procédure `gestion_graph` mais coupe simplement la liaison avec l'événement `leChef.augmentation_salaire`. Cette liaison pourra être rétablie par la suite en utilisant à nouveau `AddHandler`.

Événements et héritage

Dernier point à éclaircir : comment vont se comporter les événements lorsque notre classe sera sous-classée.

Les événements suivent les mêmes règles d'héritage que les propriétés et méthodes. Un événement déclaré dans une classe sera accessible dans toutes ces sous-classes. Une petite restriction existe quand même, puisque le déclenchement de l'événement pourra n'avoir lieu que dans la classe dans laquelle il est déclaré. Pour contourner le problème, il suffit de déclarer une méthode `protected` dans la classe et simplement faire déclencher l'événement par cette méthode. Nous aurons ensuite la possibilité de déclencher l'événement dans n'importe quelle autre sous-classe, non pas en utilisant directement le `RaiseEvent`, mais en utilisant la méthode protégée qui le fera pour nous.

Par exemple, pour pouvoir déclencher l'événement dans une sous-classe de `Client` nous pourrions écrire le code suivant :

Dans la classe `Client` :

```
Protected Sub declenche_NouvelleCommande(ByVal nb as integer)
     RaiseEvent NouvelleCommande (nb)
End sub
```

Dans la sous-classe de `Client` :

```
Public Sub plusUne()
...
    declenche_NouvelleCommande(numero)
...
End Sub
```

6. Les délégués

Il peut être parfois utile de pouvoir passer comme paramètre à une fonction ou à une procédure non pas un type simple ou objet mais directement une procédure ou fonction. Par exemple, si nous voulons choisir le critère suivant lequel nous allons faire le tri de nos employés, nous devons écrire une fonction de comparaison pour chacun des critères. Lors de l'appel de la procédure de tri, nous spécifierons quelle méthode utiliser pour effectuer la comparaison.

a. Déclaration et création d'un délégué

La première étape consiste à déclarer le délégué. Cette déclaration utilise le mot clé `Delegate` suivi de la signature de la procédure ou fonction.

```
Public Delegate Function comparaison(ByVal o1 As Object, ByVal o2 As Object)
As Integer
```

Il faut ensuite créer une procédure ou fonction qui respecte la signature du délégué. Dans notre cas, nous allons créer deux fonctions capables de comparer deux Clients.

Nous en créerons une pour effectuer la comparaison sur le code et une deuxième pour la comparaison sur le nombre de commandes. Ces fonctions peuvent être écrites dans un module ou dans une classe. Dans ce dernier cas, la fonction doit être déclarée partagée, avec le mot clé `shared` pour qu'elle soit accessible sans avoir à créer d'instance de la classe.

```
Function compareCode(ByVal c1 As Client, ByVal c2 As Client) As Integer
        Select Case c1.code
            Case Is < c2.code
                Return -1
            Case c2.code
                Return 0
            Case Else
                Return 1
        End Select
End Function

Function compareNbCommandes(ByVal c1 As Client, ByVal c2 As Client) As Integer
        Select Case c1.nbcommande
            Case Is < c2.nbcommande
                Return -1
            Case c2.nbcommande
                Return 0
            Case Else
                Return 1
        End Select
End Function
```

Il faut maintenant modifier notre fonction de tri pour qu'en plus du tableau à trier, elle prenne, comme paramètre, la fonction utilisée pour la comparaison de deux éléments du tableau. Notre fonction de tri devient donc :

```
   Public Sub tri(ByVal tablo() As Object, ByVal comparateur As comparaison)
        Dim i As Integer
        Dim j As Integer
        Dim o As Object
        For i = 0 To UBound(tablo) - 1
            For j = i + 1 To UBound(tablo)
                If comparateur.Invoke(tablo(j), tablo(i)) < 0 Then
                    o = tablo(j)
                    tablo(j) = tablo(i)
                    tablo(i) = o
                End If
            Next
        Next
    End Sub
```

Deux modifications dans notre code :

- l'ajout du paramètre comparateur comme délégué pour notre fonction de tri,
- l'utilisation de ce délégué pour comparer deux éléments du tableau en utilisant la méthode `invoke`.

Les éléments du tableau ne sont plus obligés d'implémenter l'interface `Comparable` car un moyen de les comparer deux à deux est passée à la fonction.

b. Utilisation des délégués

Pour utiliser la fonction de tri, nous devons maintenant lui fournir deux paramètres : le tableau à trier et une instance d'un délégué utilisé pour trier le tableau. Nous devons donc créer une instance d'un délégué.

```
Dim fct As comparaison
fct = New comparaison(AddressOf compareCode)
```

Le constructeur attend un paramètre lui permettant de trouver le code de la fonction. L'opérator`AddressOf` extrait l'adresse mémoire de la fonction et la passe donc au constructeur.

Nous pouvons ensuite appeler la fonction de tri en lui passant comme deuxième paramètre l'instance du délégué.

```
tri(tab, fct)
```

Si nous souhaitons utiliser un autre critère de tri, nous devons simplement créer une autre fonction respectant la signature du délégué et construire un délégué à partir de cette fonction.

c. Expressions lambda

Une expression lambda est comparable à une fonction sans nom effectuant un traitement et retournant une simple valeur. Elles peuvent être utilisées partout où un délégué est attendu.

Les types génériques

Les types génériques sont des éléments d'un programme qui s'adaptent, automatiquement, pour réaliser la même fonctionnalité sur différents types de données. Lorsque vous créez un élément générique, vous n'avez pas besoin de concevoir une version différente pour chaque type de donnée avec lequel vous souhaitez réaliser une fonctionnalité.

Pour faire une analogie avec un objet courant, nous allons prendre l'exemple d'un tournevis. En fonction du type de vis à utiliser, vous pouvez prendre un tournevis spécifique pour ce type de vis (plat, cruciforme, torx...). Une technique fréquemment utilisée par un bricoleur averti consiste à acquérir un tournevis universel avec de multiples embouts. En fonction du type de vis, il choisit l'embout adapté. Le résultat final est le même que s'il dispose d'une multitude de tournevis différents : Il peut visser et dévisser.

Lorsque vous utilisez un type générique, vous le paramétrez avec un type de données. Ceci permet au code de s'adapter automatiquement et de réaliser la même action indépendamment du type de données. Une alternative pourrait être l'utilisation du type universel `Object`. L'utilisation des types génériques présente plusieurs avantages par rapport à cette solution :

- Elle impose la vérification des types de données au moment de la compilation et évite les vérifications qui doivent être effectuées manuellement avec l'utilisation du type `Object`.
- Elle évite les opérations de conversion, du type `Object` vers un type plus spécifique et inversement, consommatrices de ressources.
- Elle évite l'utilisation de la liaison `tardive`, incontournable avec le type `Object`.
- L'écriture du code est facilitée par l'environnement de développement grâce à IntelliSense.
- Elle favorise l'écriture d'algorithmes indépendants des types de données.

Les types génériques peuvent cependant imposer certaines restrictions concernant le type de donnée utilisé. Ils peuvent, par exemple, imposer que le type utilisé implémente une ou plusieurs interfaces, qu'il soit un type référence ou possède un constructeur par défaut.

Il est important de bien comprendre quelques termes utilisés avec les génériques :

Le type générique

C'est la définition d'une classe, structure, interface ou procédure pour laquelle vous spécifiez au moins un type de données, au moment de sa déclaration.

Le type paramètre

C'est l'emplacement réservé pour le type de données dans la déclaration du type générique.

Le type argument

C'est le type de données qui remplace le type de paramètre, lors de la construction d'un type à partir d'un type générique.

Les contraintes

Ce sont les conditions que vous imposez qui limitent le type argument que vous pouvez fournir.

Le type construit

C'est la classe, interface, structure ou procédure déclarée à partir d'un type générique pour lequel vous avez spécifié des types argument.

1. Les classes génériques

Une classe qui attend un type de paramètre est appelée classe générique. Vous pouvez générer une classe construite en fournissant à la classe générique un type argument pour chacun de ces types paramètre.

a. Définition d'une classe générique

Vous pouvez définir une classe générique qui fournit les mêmes fonctionnalités sur différents types de données. Pour cela, vous devez fournir un ou plusieurs types de paramètre dans la définition de la classe. Prenons l'exemple d'une classe, capable de gérer une liste d'éléments avec les fonctionnalités suivantes :

- Ajouter un élément ;
- Supprimer un élément ;
- Se déplacer sur le premier élément ;
- Se déplacer sur le dernier élément ;
- Se déplacer sur l'élément suivant ;
- Se déplacer sur l'élément précédent ;
- Obtenir le nombre d'éléments.

Nous devons tout d'abord définir la classe comme une classe ordinaire.

```
Public Class ListeGenerique

End Class
```

La transformation de cette classe en classe générique s'effectue en ajoutant un type de paramètre immédiatement après le nom de la classe.

```
Public Class ListeGenerique(Of typeDeDonnee)

End Class
```

Si plusieurs types de paramètres sont nécessaires, il doivent être séparés par des virgules sans répéter le mot clé `Of`.

Si votre code doit réaliser d'autres opérations que des affectations, vous devez ajouter des contraintes sur le type de paramètre. Pour cela, ajouter le mot clé `As` suivi de la contrainte. Par exemple, si le paramètre doit implémenter une interface particulière, utilisez la syntaxe suivante :

```
Public Class ListeGenerique(Of typeDeDonnee As ICloneable)

End Class
```

S'il n'y a pas de contrainte de spécifiée, les seules opérations autorisées seront celles supportées par le type `Object`.

Dans le code de la classe, chaque membre qui doit être du type du paramètre doit être défini avec la syntaxe `As typeDeDonnee`, dans notre cas. Voyons maintenant le code complet de la classe.

```
Public Class ListeGenerique(Of typeDeDonnee)
    ' tableau pour stocker les éléments de la liste
    Private liste() As typeDeDonnee
    ' pointeur de position dans la liste
    Private position = 0
    ' pointeur pour l'ajout d'un nouvel element
    Private elementSuivant = 0
    'nombre d'élémments de la liste
    Private nbElements = 0
    ' dimension de la liste
    Private taille As Integer
   ' indique si la liste est pleine
    Private complet As Boolean = False
    ' constructeur avec un parametre permettant de dimensionner la liste
    Public Sub New(ByVal taille As Integer)
        liste = New typeDeDonnee(taille - 1) {}
        MyClass.taille = taille
```

```
End Sub
Public Sub ajout(ByVal element As typeDeDonnee)
    ' on verifie si la liste est complete avant
    ' d'ajouter un element
    If Not complet Then
        liste(elementSuivant) = element
        nbElements = nbElements + 1
        complet = (nbElements = taille)
        ' si la liste n'est pas complete on positionne le pointeur
        ' pour l'ajout de l'element suivant
        If Not complet Then
            elementSuivant = elementSuivant + 1
        End If
    Else
        Beep()
    End If
End Sub
Public Sub supprime(ByVal index As Integer)
    Dim i As Integer
    ' on verifie si l'index n'est pas superieur au nombre d'elements
    ' si l'index n'est pas inferieur à 0
    If index<+>>= nbElements OrElse index < 0 Then
        Beep()
        Exit Sub
    End If
    ' on decale les elements d'une position vers le haut

    For i = index To nbElements - 2
        liste(i) = liste(i + 1)
    Next
    ' on positionne le pointeur pour l'ajout d'un nouvel élément
    elementSuivant = elementSuivant - 1
    ' on met a jour le nombre d'elements
    nbElements = nbElements - 1
End Sub
Public ReadOnly Property tailleListe() As Integer
    Get
        Return nbElements
    End Get
End Property

Public Function premier() As typeDeDonnee
    If nbElements = 0 Then
        Err.Raise(1000, , "liste vide")
    End If
    ' on deplace le pointeur sur le premier element
    position = 0
    Return liste(0)
End Function
Public Function dernier() As typeDeDonnee
    If nbElements = 0 Then
        Err.Raise(1000, , "liste vide")
    End If
    ' on deplace le pointeur sur le dernier element
    position = nbElements - 1
    Return liste(position)
End Function
Public Function suivant() As typeDeDonnee
    If nbElements = 0 Then
        Err.Raise(1000, , "liste vide")
    End If
    ' on verifie si on n'est pas a la fin de la liste
    If position = nbElements - 1 Then
        Beep()
        Err.Raise(1000, , "pas d'element suivant")
        Exit Function
    End If
    ' on deplace le pointeur sur l'element suivant
    position = position + 1
```

```
        Return liste(position)
    End Function
    Public Function precedent() As typeDeDonnee
        If nbElements = 0 Then
            Err.Raise(1000, , "liste vide")
        End If
        ' on verifie si on n'est pas sur le premier element
        If position = 0 Then
            Beep()
            Err.Raise(1000, , "pas d'element precedent")
            Exit Function
        End If
        ' on se deplace sur l'element precedent
        position = position - 1
        Return liste(position)
    End Function
  End Class
```

b. Utilisation d'une classe générique

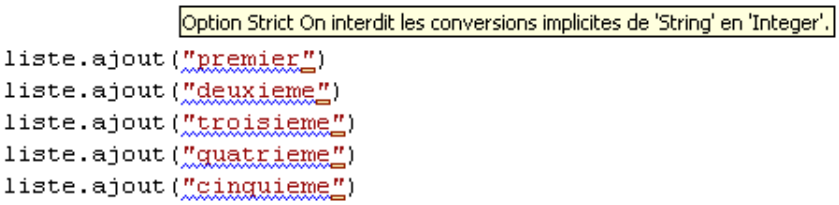
Pour pouvoir utiliser une classe générique, vous devez tout d'abord générer une classe construite en fournissant un type argument pour chacun de ces types paramètre. Vous pouvez alors instancier la classe construite par un des constructeurs disponibles. Nous allons utiliser la classe conçue précédemment pour travailler avec une liste d'entiers.

```
Dim liste As New ListeGenerique(Of Integer)(5)
```

Cette déclaration permet d'instancier une liste de cinq entiers. Les méthodes de la classe sont alors disponibles.

```
liste.ajout(10)
liste.ajout(11)
```

Le compilateur vérifie bien sûr que nous utilisons notre classe correctement, notamment en vérifiant les types de données que nous lui confions.



Voici le code d'une petite application permettant de tester le bon fonctionnement de notre classe générique :

```
Module testGenerique
    Dim liste As New ListeGenerique(Of Integer)(5)
    Public Sub main()
        liste.ajout(10)
        liste.ajout(11)
        liste.ajout(12)
        liste.ajout(13)
        liste.ajout(14)
        liste.ajout(15)
        menu()
    End Sub
    Public Sub menu()
        Dim choix As Char
        On Error GoTo gestionerreur
        Console.SetCursorPosition(1, 24)
        Console.WriteLine("p (premier) < (precedent) >(suivant) d (dernier) f
(fin)")
        While choix <> "f"
            choix = Console.ReadLine()
          Console.Clear()
            Console.SetCursorPosition(1, 1)
            Select Case choix
                Case "p"
```

```
                    Console.WriteLine("le premier {0}", liste.premier())
                Case "<"
                    Console.WriteLine("le precedent {0}", liste.precedent())
                Case ">"
                    Console.WriteLine("le suivant {0}", liste.suivant())
                Case "d"
                    Console.WriteLine("le dernier {0}", liste.dernier())
            End Select
            Console.SetCursorPosition(1, 24)
            Console.WriteLine("p (premier) < (precedent) >(suivant) d
(dernier) f (fin)")
        End While
        Exit Sub
gestionerreur:
        Console.ForegroundColor = ConsoleColor.Red
        Console.WriteLine(Err.Description)
        Console.ResetColor()
        Resume Next
    End Sub
End Module
```

Nous pouvons également vérifier que notre classe fonctionne sans problème si nous lui demandons de travailler avec des chaînes de caractères.

```
    Public Sub main()
        liste.ajout("premier")
        liste.ajout("deuxieme")
        liste.ajout("troisieme")
        liste.ajout("quatrieme")
        liste.ajout("cinquieme")
        menu()
    End Sub
```

2. Procédures et fonctions génériques

Une procédure ou une fonction générique sont des méthodes définies avec au moins un type paramètre. Ceci permet au code appelant de spécifier le type de données dont il a besoin à chaque appel de la procédure ou fonction. Une telle méthode peut cependant être utilisée sans indiquer d'information pour le type argument. Dans ce cas, le compilateur essaie de déterminer le type, en fonction des arguments passés à la méthode. Cette solution doit toutefois être utilisée avec précaution, car si le compilateur ne peut pas déterminer le type des arguments, il génère une erreur de compilation.

a. Création d'une procédure ou fonction générique

La déclaration d'une procédure ou fonction générique doit contenir au moins un type paramètre. Ce type paramètre est défini par l'utilisation du mot clé `Of` suivi d'un identifiant. Cet identifiant est ensuite utilisé dans le reste du code, à chaque fois que vous avez besoin d'utiliser le type paramètre.

Nous allons créer une fonction générique capable de rechercher un élément particulier dans un tableau de n'importe quel type. Cette fonction va utiliser un type paramètre indiquant la nature des éléments présents dans le tableau. Pour pouvoir rechercher un élément dans le tableau, nous devrons le comparer avec ceux présents dans toutes les cases du tableau. Pour s'assurer que cette comparaison sera possible, nous ajoutons une contrainte sur le type paramètre : il doit implémenter l'interface `Icomparable` afin de garantir que la méthode `CompareTo` utilisée dans la fonction soit disponible pour chaque élément du tableau. La déclaration de la fonction prend la forme suivante :

```
Public function rechercheGenerique(Of typeDonnee As Icomparable) (byval
tablo As typeDonne(), elementRecherche As typeDonnee) As Integer
```

Après avoir vérifié que le tableau contient au moins un élément, nous devons comparer l'élément recherché avec celui présent dans chaque case du tableau. S'il y a égalité, la fonction retourne l'index où l'élément a été trouvé, sinon la fonction retourne -1. Pour effectuer la comparaison, nous utiliserons la fonction `CompareTo` de chaque élément du tableau.

```
Public Function rechercheGenerique(Of typeDonnee As Icomparable)(ByVal tablo
As typeDonnee(), ByVal elementRecherche As typeDonnee) As Integer
        'test si le tableau a plus d'une dimension
        If tablo.Rank() > 1 Then
```

```
            Return -1
        End If
        ' test si le tableau est vide
        If tablo.Length = 0 Then
            Return -1
        End If
        For i As Integer = 0 To tablo.GetUpperBound(0)
            If tablo(i).CompareTo(elementRecherche) = 0 Then
                Return i
            End If
        Next
        Return -1
    End Function
```

b. Utilisation d'une procédure ou fonction générique

L'utilisation d'une procédure ou fonction générique est identique à celle d'une procédure ou fonction classique, hormis la nécessité de spécifier un type argument pour le ou les types paramètre.

Le code suivant permet de tester le bon fonctionnement de notre fonction.

```
Public Sub main()
        Dim t() As Integer = {12, 45, 85, 47, 62, 95, 81}
        Dim resultat As Integer
        resultat = rechercheGenerique(Of Integer)(t, 47)
        If resultat = -1 Then
            Console.WriteLine("valeur non trouvée")
        Else
            Console.WriteLine("valeur trouvée à la position {0}", resultat)
        End If
        Console.ReadLine()
        Dim s() As String = {"un", "deux", "trois", "quatre", "cinq"}
        resultat = rechercheGenerique(Of String)(s, "six")
        If resultat = -1 Then
            Console.WriteLine("valeur non trouvée")
        Else
            Console.WriteLine("valeur trouvée à la position {0}", resultat)
        End If
        Console.ReadLine()
    End Sub
```

Les collections

Les applications ont très fréquemment besoin de manipuler de grandes quantités d'information. De nombreuses structures sont disponibles en Visual Basic pour faciliter la gestion de ces informations. Elles sont regroupées sous le terme `collection`. Comme dans la vie courante, il y a différents types de collection. Il peut y avoir des personnes qui récupèrent toute sorte de choses mais qui n'ont pas d'organisation particulière pour les ranger, d'autres qui sont spécialisées dans la collection de certains types d'objets, les maniaques qui prennent toutes les précautions possibles pour pouvoir retrouver à coup sûr un objet...

Il existe dans le Framework .NET des classes correspondant à chacune de ces situations.

1. Les collections prédéfinies

Les différentes classes permettant la gestion de collections sont réparties dans deux espaces de nom :

- System.Collections
- System.Collections.Generic

Le premier contient les classes normales alors que le deuxième contient les classes génériques équivalentes permettant la création de collections fortement typées. Ces collections fortement typées sont spécialisées dans la gestion d'un type précis de données. Bien que ces nombreuses classes fournissent des fonctionnalités différentes, elles ont quand même des points communs, dûs au fait qu'elles implémentent les mêmes interfaces. Par exemple, toutes ces classes sont capables de fournir un objet `enumerator` permettant de parcourir l'ensemble de la collection. C'est cet objet qui est d'ailleurs utilisé par l'instruction `For Each` de Visual Basic.

a. Array

La classe `Array` ne fait pas partie de l'espace de nom `System.Collections` mais elle peut quand même être considérée comme une collection car elle implémente l'interface `Ilist`. Les tableaux créés à partir de la classe `Array` sont de taille fixe, même s'il est possible d'utiliser l'instruction `redim` pour redimensionner un tableau. En fait, l'instruction `redim` n'agrandit pas le tableau mais crée un nouveau tableau plus grand ou plus petit et recopie éventuellement les valeurs existantes, si le mot clé `Preserve` est utilisé. Cette classe contient également une multitude de méthodes partagées, permettant l'exécution de nombreuses fonctionnalités sur des tableaux. Deux propriétés sont très utiles pour l'utilisation de la classe `Array` :

- `Length` qui représente le nombre total d'éléments dans le tableau.
- `Rank` qui contient le nombre de dimensions du tableau.

Cette classe est rarement utilisée pour la création d'un tableau car l'on préfère utiliser la syntaxe Visual Basic pour cela.

b. ArrayList et List

La classe `ArrayList` ou sa version générique `List` sont des évolutions de la classe `Array`. Elles apportent de nombreuses améliorations par rapport à cette dernière.

- La taille d'un `ArrayList` est dynamique et est automatiquement ajustée en fonction des besoins. Le traitement, qui pour un `Array` devait être réalisé par l'instruction `Redim`, est effectué automatiquement.
- Elle propose des méthodes permettant l'ajout, l'insertion et la suppression de plusieurs éléments simultanément en une seule opération.

Par contre, sur certains points, la classe `ArrayList` est moins efficace qu'un simple tableau :

- Les `ArrayList` n'ont qu'une seule dimension.
- Un tableau de données d'un type spécifique est plus efficace qu'un `ArrayList` dont les éléments sont gérés

en tant qu'Object. L'utilisation de la version générique (List) permet d'obtenir des performances équivalentes.

Comme toute classe, un ArrayList doit être instancié avant de pouvoir être utilisé. Deux constructeurs sont disponibles. Le premier est le constructeur par défaut et crée un ArrayList avec une capacité initiale de zéro. Il sera ensuite dimensionné, automatiquement, lors de l'ajout d'éléments. Cette solution n'est pas conseillée car l'agrandissement de l'ArrayList consomme beaucoup de ressources. Si vous avez une estimation du nombre d'éléments à stocker, il est préférable d'utiliser le deuxième constructeur qui attend comme paramètre la capacité initiale de l'ArrayList. Ceci évite le dimensionnement automatique lors de l'ajout.

À noter que la taille indiquée n'est pas définitive et l'ArrayList pourra contenir plus d'éléments prévus initialement.

La propriété Capacity permet de connaître le nombre d'éléments que l'ArrayList peut contenir. La propriété Count indique le nombre actuel d'éléments dans l'ArrayList. Les méthodes Add et AddRange ajoutent des éléments à la fin de la liste. Les méthodes Insert et InsertRange permettent de choisir l'emplacement où va s'effectuer l'ajout. La propriété Item, qui est la propriété par défaut de classe, s'utilise pour atteindre un élément à une position donnée. La suppression d'éléments se fait par la méthode RemoveAt ou RemoveRange ; la première attend comme paramètre l'index de l'élément à supprimer, la deuxième exige en plus le nombre d'éléments à supprimer. La méthode Clear est plus radicale et supprime tous les éléments.

Le code suivant illustre le fonctionnement de cette classe :

```
Public Sub main()
        Dim liste As ArrayList
        Dim c As Client
        liste = New ArrayList()
        Console.WriteLine("capacité initiale de la liste {0}", liste.Capacity)
        Console.WriteLine("nombre d'éléments de la liste {0}", liste.Count)
        Console.WriteLine("ajout d'un client")
        c = New Client("client1", "prenom1", "secret", 1001)
        liste.Add(c)
        Console.WriteLine("capacité de la liste {0}", liste.Capacity)
        Console.WriteLine("nombre d'éléments de la liste {0}", liste.Count)
        Console.WriteLine("ajout de quatre clients")
        c = New Client("client2", "prenom2", "secret", 1002)
        liste.Add(c)
        c = New Client("client3", "prenom3", "secret", 1003)
        liste.Add(c)
        c = New Client("client4", "prenom4", "secret", 1004)
        liste.Add(c)
        c = New Client("client5", "prenom5", "secret", 1005)
        liste.Add(c)
        Console.WriteLine("capacité de la liste {0}", liste.Capacity)
        Console.WriteLine("nombre d'éléments de la liste {0}", liste.Count)
        Console.WriteLine("affichage de la liste des clients")
        For Each c In liste
            c.affichage()
            Console.WriteLine()
        Next
        Console.WriteLine("effacement des clients 1002, 1003, 1004")
        liste.RemoveRange(1, 3)
        Console.WriteLine("capacité de la liste {0}", liste.Capacity)
        Console.WriteLine("nombre d'éléments de la liste {0}", liste.Count)
        Console.WriteLine("affichage de la liste des clients")
        For Each c In liste
            c.affichage()
            Console.WriteLine()
        Next
        Console.WriteLine("affichage du deuxième client de la liste")
        liste(1).affichage()
        Console.WriteLine()
        Console.WriteLine("effacement de  tous les clients")
        liste.Clear()
        Console.WriteLine("capacité de la liste {0}", liste.Capacity)
        Console.WriteLine("nombre d'éléments de la liste {0}", liste.Count)
        Console.ReadLine()
    End Sub
```

Il affiche le résultat suivant :

```
capacité initiale de la liste 0
nombre d'éléments de la liste 0
ajout d'un client
capacité de la liste 4
nombre d'éléments de la liste 1
ajout de quatre clients
capacité de la liste 8
nombre d'éléments de la liste 5
affichage de la liste des clients
Mr client1 prenom1 né le 01/01/0001 00:00:00 code Client : 1001
Mr client2 prenom2 né le 01/01/0001 00:00:00 code Client : 1002
Mr client3 prenom3 né le 01/01/0001 00:00:00 code Client : 1003
Mr client4 prenom4 né le 01/01/0001 00:00:00 code Client : 1004
Effacement des clients 1002, 1003, 1004
capacité de la liste 8
nombre d'éléments de la liste 2
affichage de la liste des clients
Mr client1 prenom1 né le 01/01/0001 00:00:00 code Client : 1001
Mr client5 prenom5 né le 01/01/0001 00:00:00 code Client : 1005
affichage du deuxième client de la liste
Mr client5 prenom5 né le 01/01/0001 00:00:00 code Client : 1005
Effacement de  tous les clients
capacité de la liste 8
nombre d'éléments de la liste 0
```

La capacité de la liste ne diminue pas lors de la suppression d'un élément, même lorsque la liste est vide.

c. Hashtable et Dictionary

Une `Hashtable` ou sa version générique `Dictionary` enregistre les informations sous forme de couple clés valeur. La `Hashtable` est constituée en interne de compartiments contenant les éléments de la collection. Pour chaque élément de la collection, un code est généré par une fonction de hachage, basée sur la clé de chaque élément. Le code est ensuite utilisé pour identifier le compartiment dans lequel est stocké l'élément. Lors de la recherche d'un élément dans la collection, l'opération inverse est effectuée. Le code de hachage est généré à partir de la clé de l'élément recherché. Cette clé sert ensuite à identifier le compartiment dans lequel se trouve l'élément recherché. Pour qu'une `Hashtable` puisse stocker un objet, celui-ci doit être capable de fournir son propre code de hachage.

d. Queue

Ce type de collection est utilisé lorsque vous avez besoin d'un espace de stockage temporaire. Lorsqu'un élément est récupéré à partir de la collection, il est en même temps supprimer de la collection.

Les collections de type `Queue` sont adaptées, si vous avez besoin d'accéder aux informations dans le même ordre que celui dans lequel elles ont été stockées dans la collection. Ce type de gestion est parfois appelé First In - First Out (FiFo). Les trois principales opérations disponibles sont :

- `Enqueue` pour ajouter un élément à la fin de la queue,
- `Dequeue` pour obtenir l'élément le plus ancien de la queue et le supprimer,
- `Peek` pour obtenir l'élément le plus ancien sans le supprimer de la queue.

L'exemple suivant illustre l'utilisation de ces trois méthodes.

```
  Public Sub main()
       Dim q As Queue
       q = New Queue
       Dim c As Client
       c = New Client("client1", "prenom1", "secret", 1001)
       Console.WriteLine("arrivée du premier client:{0}", c.nom)
       q.Enqueue(c)
```

```
        c = New Client("client2", "prenom2", "secret", 1002)
        Console.WriteLine("arrivée du deuxième client:{0}", c.nom)
        q.Enqueue(c)
        c = New Client("client3", "prenom3", "secret", 1003)
        Console.WriteLine("arrivée du troisième client:{0}", c.nom)
        q.Enqueue(c)
        Console.WriteLine("départ du premier client:{0}", q.Dequeue.nom)
        Console.WriteLine("il reste {0} clients", q.Count)
        Console.WriteLine("départ du deuxième client:{0}", q.Dequeue.nom)
        Console.WriteLine("il reste {0} client", q.Count)
        Console.WriteLine("le troisième client s'incruste:{0}", q.Peek.nom)
        Console.WriteLine("il reste {0} client", q.Count)
        Console.WriteLine("départ du troisième client:{0}", q.Dequeue.nom)
        Console.WriteLine("il reste {0} client", q.Count)
        Console.ReadLine()
    End Sub
```

e. Stack

Les collections de ce type utilisent le même principe que les `Queue` : lorsqu'un élément est récupéré de la collection, il en est supprimé. La seule distinction par rapport à la classe `Queue` est l'ordre dans lequel les éléments sont récupérés. Ce type de collection utilise la technique `last in - first out` (lifo). L'exemple classique de ce type de gestion est la pile d'assiettes de votre cuisine. Après avoir fait la vaisselle, vous empilez les assiettes sur une étagère. Le lendemain lorsque vous mettez le couvert, la première assiette disponible est la dernière, rangée la veille.

Les trois principales opérations disponibles sont :

- `Push` pour ajouter un élément au sommet de la pile ;
- `Pop` pour obtenir l'élément au sommet de la pile et le supprimer ;
- `Peek` pour obtenir l'élément au sommet de la pile sans le supprimer de la pile.

2. Choisir un type de collection

Voici quelques conseils pour choisir le type de collection adapté à vos besoins.

- Vous avez besoin d'accéder aux éléments de la collection par un index : utilisez une `ArrayList`.
- L'accès aux éléments doit s'effectuer dans l'ordre de l'ajout dans la collection ou dans l'ordre inverse : utilisez une `Queue` ou une `Stack`.
- Vous avez besoin de trier les éléments dans un ordre différent de celui dans lequel ils sont ajoutés à la collection : utilisez une `ArrayList` ou une `Hashtable`.
- Les éléments à stocker dans la liste sont des couples clé-élément : utilisez une `Hashtable`.

Les objets intrinsèques

De nombreux objets sont disponibles automatiquement à partir de Visual Basic sans que vous ayez à en créer une instance. Ces objets sont accessibles par le mot clé`My`. Ils permettent la manipulation et l'accès à des informations, fréquemment utilisées au cours du fonctionnement d'une application. Ils fournissent, par exemple, un moyen d'accéder aux propriétés de la machine sur laquelle s'exécute l'application, par l'intermédiaire de `My.Computer`, ou aux propriétés de l'application elle-même par l'intermédiaire de `My.Application`. En fonction du type de projet sur lequel vous travaillez, il est possible que certains objets ne soient pas disponibles. Le tableau suivant présente les objets existants et le contexte dans lequel ils sont disponibles.

	Application Windows	**Bibliothèque de classes**	**Application Console**	**Bibliothèque de contrôles Windows**	**Service Windows**
My.Application	Oui	Oui	Oui	Oui	Oui
My.Computer	Oui	Oui	Oui	Oui	Oui
My.Forms	Oui	Non	Non	Oui	Non
My.Resources	Oui	Oui	Oui	Oui	Oui
My.Settings	Oui	Oui	Oui	Oui	Oui
My.User	Oui	Oui	Oui	Oui	Oui
My.WebServices	Oui	Oui	Oui	Oui	Oui

En fonction du type de projet dans lequel ils sont utilisés, certaines propriétés de ces objets peuvent être indisponibles. Par exemple, la propriété `MainForm` de l'objet `Application` n'est utilisable que dans les projets d'application Windows.

- L'objet `Application` fournit des propriétés méthodes et événements concernant l'application courante. Il permet, par exemple, la récupération d'informations sur la configuration linguistique de l'application, les paramètres de la ligne de commande utilisés pour lancer l'application ou encore les informations concernant la version de l'application.

- L'objet `Computer` facilite l'accès aux différentes ressources, accessibles sur la machine. Il permet par exemple l'accès direct au système audio de la machine, au clavier, à la souris ou encore au réseau.

- L'objet `Forms` met à votre disposition une instance de chacune des fenêtres disponibles dans le projet. L'accès se fait par le nom de classe correspondant à la fenêtre qui devient ainsi une propriété de l'objet `Forms`.

- L'objet `Resources` permet l'accès aux ressources audio, icônes, images et chaînes de caractères définies au niveau du projet.

- L'objet `Settings` permet l'accès aux paramètres de l'application définis par les propriétés du projet.

- L'objet `User` représente l'utilisateur de l'application. Si aucune authentification spécifique n'est utilisée dans l'application alors l'objet `User` correspond à l'utilisateur avec lequel la session Windows a été ouverte.

- L'objet `Webservices` fournit une instance de chaque service web référencé dans l'application. Comme pour la propriété `Forms`, l'accès s'effectue par le nom du service Web.

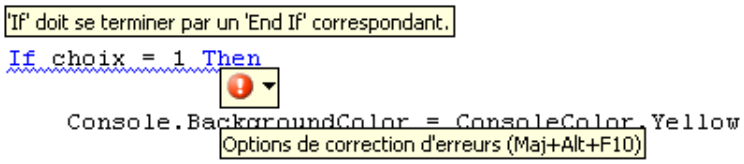
Les différents types d'erreurs

Pour un développeur, les erreurs sont une des principales sources de stress. En fait, nous pouvons classer ces erreurs en trois catégories. Regardons chacune d'entre elles et les solutions disponibles pour les traiter.

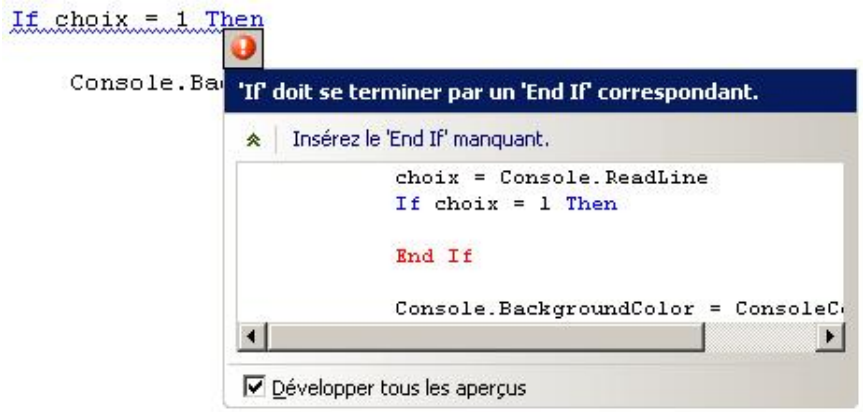
1. Les erreurs de syntaxe

Ce type d'erreur se produit au moment de la compilation, lorsqu'un mot clé du langage est mal orthographié. Très fréquentes avec les premiers outils de développement où l'éditeur de code et le compilateur étaient deux entités séparées, elles deviennent de plus en plus rares avec les environnements tels que Visual Studio. La plupart de ces environnements proposent une analyse syntaxique au fur et à mesure de la saisie du code. De ce point de vue, Visual Studio propose de nombreuses fonctionnalités nous permettant d'éliminer ces erreurs.

Il surveille, par exemple, que chaque instruction `If` est bien terminée par un `End If`.

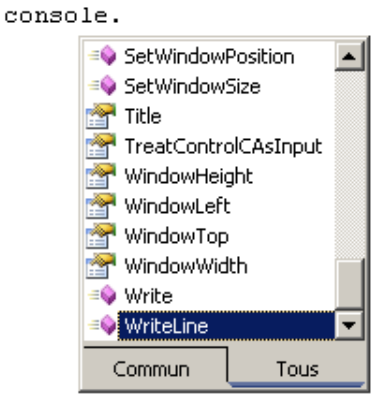


Si une erreur de syntaxe est détectée, Visual Basic propose les solutions possibles pour corriger cette erreur. Elles sont affichées en cliquant sur l'icône associée à l'erreur.

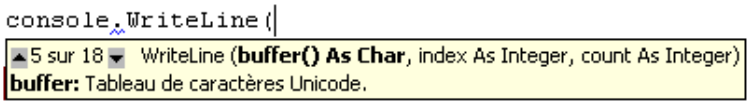


D'autre part les "fautes d'orthographe" dans les noms de propriétés ou de méthodes sont facilement éliminées grâce aux fonctionnalités `IntelliSense`. `IntelliSense` prend en charge les fonctionnalités suivantes :

- L'affichage automatique de la liste des membres disponibles :

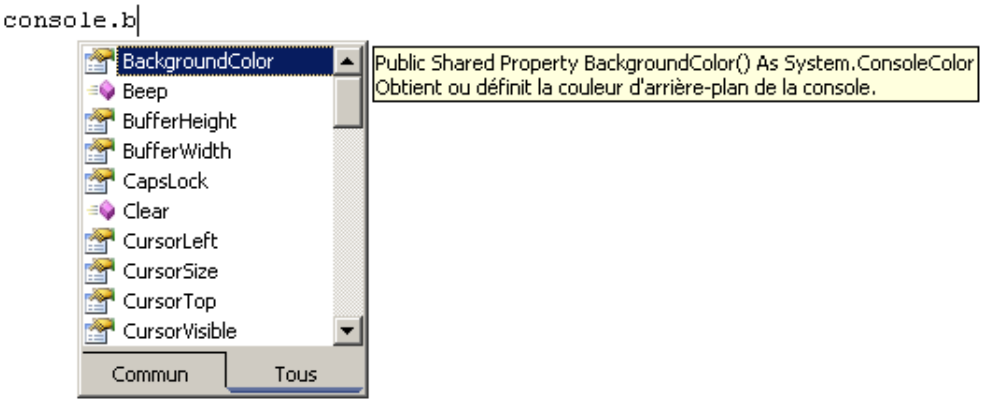


- L'affichage de la liste des paramètres à fournir pour l'appel d'une procédure ou fonction :



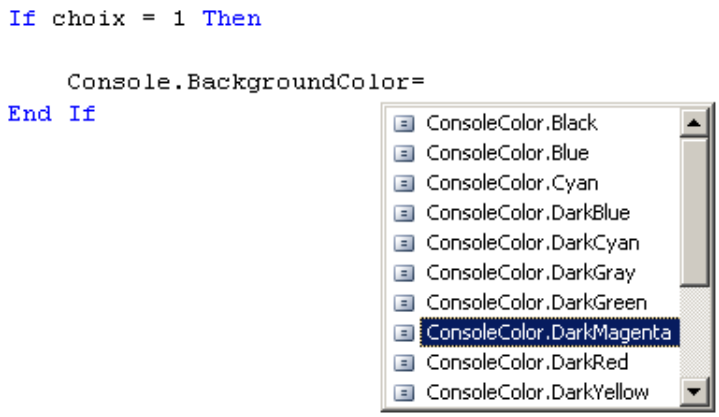
Si plusieurs surcharges sont disponibles, IntelliSense affiche leur nombre et vous permet de les parcourir en utilisant les flèches haut et bas du clavier.

- L'affichage d'informations ponctuelles sur membres d'une classe :



- Le complément automatique de mots : commencez à saisir un début de mot puis utilisez la combinaison de touches [Ctrl] [Espace] pour afficher tout ce que vous pouvez utiliser comme mot, à cet emplacement, commençant par les caractères déjà saisis. S'il n'y a qu'une seule possibilité, le mot est ajouté automatiquement, sinon sélectionnez-le dans la liste et validez avec la touche [Tab].

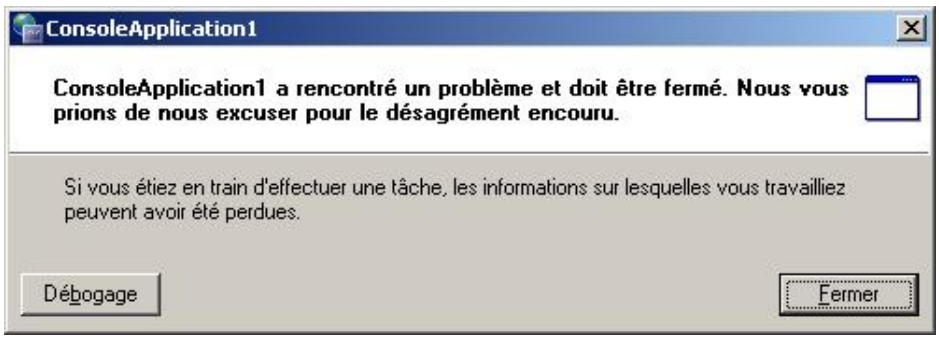
- L'affichage de la liste des valeurs possibles pour une propriété de type énumération.



Avec toutes ces fonctionnalités, il est pratiquement impossible de générer des fautes de syntaxe dans le code.

2. Les erreurs d'exécution

Ces erreurs apparaissent après la compilation, lorsque vous lancez l'exécution de votre application. La syntaxe du code est correcte mais l'environnement de votre application ne permet pas l'exécution d'une instruction utilisée dans votre application. C'est, par exemple, le cas si vous essayez d'ouvrir un fichier qui n'existe pas sur le disque de votre machine. Vous obtiendrez sûrement une boîte de dialogue de ce type.



Ce type de boîte de dialogue n'est pas très sympathique pour l'utilisateur !

Heureusement, Visual Basic permet la récupération de ce type d'erreur et évite ainsi l'affichage de cette inquiétante boîte de dialogue. Deux techniques sont disponibles pour la gestion de ce type d'erreurs :

- la gestion en ligne ;
- les exceptions.

Nous détaillerons cela un peu plus loin dans ce chapitre.

Les erreurs de logique

Les pires ennemis des développeurs. Tout se compile sans problème, tout s'exécute sans problème et pourtant "ça ne marche pas" !!!

Il convient, dans ce cas, de revoir la logique de fonctionnement de l'application. Les outils de débogage nous permettent de suivre le déroulement de l'application, de placer des points d'arrêt, de visualiser le contenu des variables, etc.

Traitement des erreurs

Deux techniques sont disponibles pour le traitement des erreurs dans Visual Basic :

- la gestion en ligne ;
- le traitement par exception.

1. La gestion en ligne

L'instruction `On Error` est l'élément de base de la gestion des erreurs en ligne. Lorsque cette instruction est exécutée, dans une procédure ou fonction, elle active la gestion des erreurs pour cette procédure ou fonction. Il convient cependant d'indiquer comment notre gestionnaire doit réagir lorsqu'une instruction déclenche une erreur. Deux solutions :

```
On error resume next
```

L'exécution du code va se poursuivre par la ligne suivant celle qui a provoqué l'erreur.

```
On error goto etiquette
```

L'exécution du code va se poursuivre par la ligne repérée par `etiquette`.

La syntaxe est donc la suivante :

```
Private Sub Nom_de_procedure()
   On Error Goto gestionErreurs
      ...
      "Instructions dangereuses"
  ...
Exit Sub
gestionErreurs:
  ...
  code de gestion des erreurs
  ...
End Sub
```

L'étiquette vers laquelle le gestionnaire d'erreurs va rediriger l'exécution doit se trouver dans la même procédure que l'instruction `on error goto`. L'instruction `Exit sub` est obligatoire pour que le code de gestion d'erreurs ne soit pas exécuté à la suite des instructions normales mais seulement en cas d'erreur.

Le code du gestionnaire d'erreur doit déterminer la conduite à tenir en cas d'erreur.

Trois solutions :

`Resume`

On essaie à nouveau d'exécuter la ligne qui a provoqué l'erreur.

`resume next`

On continue l'exécution par la ligne suivant celle qui a provoqué l'erreur.

`exit sub` ou `exit function`

On abandonne l'exécution de cette procédure ou fonction.

Typiquement, le gestionnaire d'erreurs affichera une boîte de dialogue demandant à l'utilisateur ce qu'il souhaite.

En fonction de sa réponse, on utilisera l'une ou l'autre des solutions.

```
Private Sub OuvertureFichier()
        On Error GoTo gestionErreurs
        My.Computer.FileSystem.OpenTextFileReader("c:\essai")
        Exit Sub
```

```
gestionErreurs:
        Dim reponse As Integer
        reponse = MsgBox("impossible de lire le fichier",
MsgBoxStyle.AbortRetryIgnore)
        Select Case reponse
            Case MsgBoxResult.Retry
                Resume
            Case MsgBoxResult.Ignore
                Resume Next

            Case MsgBoxResult.Abort
                Exit Sub
        End Select
    End Sub
```

Il nous reste encore un problème à résoudre : notre gestionnaire d'erreurs réagira quelle que soit l'erreur. Pour pouvoir déterminer quelle erreur vient de se produire dans l'application, nous avons à notre disposition l'objet`err` qui nous fournit des informations sur la dernière erreur apparue. Cet objet contient, entre autres, deux propriétés, `number` et `description`, nous permettant d'obtenir le code de l'erreur et sa description. Nous pouvons donc modifier notre code pour réagir de façon différente en fonction de l'erreur.

```
Private Sub OuvertureFichierBis()
        On Error GoTo gestionErreurs
        My.Computer.FileSystem.OpenTextFileReader("a:\essai")
        Exit Sub
gestionErreurs:
        Dim reponse As Integer
        Console.WriteLine(Err.Number)
        Stop
        If Err.Number = 53 Then
            reponse = MsgBox("impossible de trouver le fichier", MsgBoxStyle.
AbortRetryIgnore)
            Select Case reponse
                Case MsgBoxResult.Retry
                    Resume
                Case MsgBoxResult.Ignore
                    Resume Next
                Case MsgBoxResult.Abort
                    Exit Sub
            End Select
        End If
        If Err.Number = 57 Then
            reponse = MsgBox("insérer une disquette dans le lecteur",
MsgBoxStyle.OKCancel)
            Select Case reponse
                Case MsgBoxResult.OK
                    Resume
                Case MsgBoxResult.Cancel
                    Exit Sub
            End Select
        End If
    End Sub
```

Un gestionnaire d'erreurs peut être désactivé en utilisant l'instruction `on error goto 0`. Si une erreur se produit après cette instruction, elle n'est pas récupérée et l'application s'arrête.

En fait, l'application ne s'arrête pas immédiatement mais Visual Basic recherche dans les fonctions appelantes si un gestionnaire d'erreur est actif et s'il en trouve un, il lui confie la gestion de l'erreur.

```
Private Sub procedure1()
        On Error GoTo gestionErreurs
        procedure2()
        Exit Sub
gestionErreurs:
        MsgBox("erreur d'execution")
    End Sub
    Private Sub procedure2()
        Dim x, y As Integer
        x = 0
        y = (1 / x)
```

```
End Sub
```

Dans cet exemple, la procedure2 va exécuter une instruction déclenchant une erreur (1 / x). Comme il n'y a pas de gestionnaire d'erreurs dans cette procédure, Visual Basic va rechercher dans la pile des appels si un gestionnaire est actif. Le premier rencontré se trouve dans la procedure1, ce sera donc celui-ci qui se chargera de la gestion de l'erreur.

2. Les exceptions

a. Récupération d'exceptions

La gestion des exceptions donne la possibilité de protéger un bloc de code contre les erreurs d'exécution qui pourraient s'y produire. Le code dangereux doit être placé dans un bloc`Try End Try`. Si une exception est déclenchée dans ce bloc de code, Visual Basic regarde les instructions`Catch` qui suivent. S'il en existe une capable de traiter l'exception, le code correspondant est exécuté sinon la même exception est déclenchée pour éventuellement être récupérée par un bloc `Try End Try`, de plus haut niveau. Une instruction `Finally`permet de marquer un groupe d'instructions, exécutées avant la sortie du bloc `Try`, qu'une erreur se soit produite ou non.

La syntaxe générale est donc la suivante :

```
Try
...
Instructions dangereuses
...
catch exception1
...
code exécuté si une exception de type Exception1 se produit
...
catch exception2
...
code exécuté si une exception de type Exception1 se produit
...
Finally
...
code exécuté dans tous les cas avant la sortie du bloc Try
...
End Try
```

Cette structure a un fonctionnement très similaire au `select case` déjà étudié. Chaque type d'erreur est associé à une classe d'exception et lorsque cette erreur se produit, une instance de la classe `exception` correspondante est créée. Nous pourrons donc déterminer, pour chaque instruction `catch`, quel type d'exception elle doit traiter.

La classe de base est la classe `Exception` à partir de laquelle est créée une multitude de sous-classes spécialisées chacune pour un type d'erreur particulier. Voici la liste des classes dérivant directement de la classe `Exception`.

- Microsoft.Build.BuildEngine.InternalLoggerException
- Microsoft.Build.BuildEngine.InvalidProjectFileException
- Microsoft.Build.Framework.LoggerException
- Microsoft.JScript.CmdLineException
- Microsoft.JScript.ParserException
- Microsoft.VisualBasic.ApplicationServices
- Microsoft.VisualBasic.ApplicationServices.NoStartupFormException
- Microsoft.VisualBasic.CompilerServices.IncompleteInitialization
- Microsoft.VisualBasic.CompilerServices.InternalErrorException

- Microsoft.VisualBasic.FileIO.MalformedLineException
- Microsoft.WindowsMobile.DirectX.DirectXException
- System.ApplicationException
- System.ComponentModel.Design.ExceptionCollection
- System.Configuration.Provider.ProviderException
- System.Configuration.SettingsPropertyCannotBeSetForAnonymousUserException
- System.Configuration.SettingsPropertyIsReadOnlyException
- System.Configuration.SettingsPropertyNotFoundException
- System.Configuration.SettingsPropertyWrongTypeException
- System.DirectoryServices.ActiveDirectory.ActiveDirectoryObjectExistsException
- System.DirectoryServices.ActiveDirectory.ActiveDirectoryObjectNotFoundException
- System.DirectoryServices.ActiveDirectory.ActiveDirectoryOperationException
- System.DirectoryServices.ActiveDirectory.ActiveDirectoryServerDownException
- System.DirectoryServices.Protocols.DirectoryException
- System.IO.IsolatedStorage.IsolatedStorageException
- System.Net.Mail.SmtpException
- System.Runtime.Remoting.MetadataServices.SUDSGeneratorException
- System.Runtime.Remoting.MetadataServices.SUDSParserException
- System.SystemException
- System.Web.Security.MembershipCreateUserException
- System.Web.Security.MembershipPasswordException
- System.Web.UI.ViewStateException
- System.Windows.Forms.AxHost.InvalidActiveXStateException

Cette liste ne présente que le premier niveau de la hiérarchie. Chacune de ces classes a elle aussi de nombreux descendants.

Ces différentes classes sont utilisées pour indiquer dans chaque instruction `Catch` le type d'exception qu'elle doit gérer.

```
   Private Sub OuvertureFichier()
       Try
           My.Computer.FileSystem.OpenTextFileReader("a:\essai")
```

```
        Catch ex As System.IO.IOException
            MsgBox("erreur d'ouverture du fichier", MsgBoxStyle.OKOnly)
        Finally
            MsgBox("fin de la procédure d'ouverture de fichier")
        End Try
    End Sub
```

Si, parmi tous les Catch, aucun ne correspond à l'exception générée, l'exception est propagée dans le code des procédures ou fonctions appelantes, à la recherche d'une instruction catch capable de prendre en compte cette exception. Si aucun bloc n'est trouvé, une erreur d'exécution est déclenchée.

Les blocs Catch peuvent également être conditionnels, en ajoutant le mot When suivi d'une expression pouvant être évaluée comme un Boolean.

```
Catch ex As Exception When choix < 0
...
Catch ex As Exception When choix > 10
  ...
End Try
```

Le bloc Catch est alors exécuté si une exception de ce type est déclenchée et si la condition est vérifiée.

Si le paramètre indiqué à l'instruction Catch est une classe "d'exception générale", cette instruction Catch sera capable de capturer toutes les exceptions créées à partir de cette classe ou de ces sous-classes. Le code suivant nous permet donc de capturer toutes les exceptions.

```
    Private Sub OuvertureFichier()
        Try
            My.Computer.FileSystem.OpenTextFileReader("a:\essai")
        Catch ex As Exception
            MsgBox("erreur d'ouverture du fichier", MsgBoxStyle.OKOnly)
        Finally
            MsgBox("fin de la procédure d'ouverture de fichier")
        End Try
    End Sub
```

Les différentes classes disposent des propriétés suivantes, nous permettant d'avoir plus d'informations sur l'origine de l'exception.

`Message`

Chaîne de caractères associée à l'exception.

`Source`

Nom de l'application qui a déclenché l'exception.

`StackTrace`

Liste de toutes les méthodes par lesquelles l'application est passée avant le déclenchement de l'erreur.

`TargetSite`

Nom de la méthode ayant déclenché l'exception.

`InnerException`

Obtient l'exception originale, si deux exceptions sont déclenchées en cascade.

b. Création et déclenchement d'exceptions

Les exceptions sont avant tout des classes, il est donc possible de créer nos propres exceptions en héritant d'une des nombreuses classes d'exception déjà disponibles. Pour respecter les conventions du Framework .Net, il est conseillé de conserver le terme Exception dans le nom de la classe. Nous pouvons par exemple écrire le code suivant :

```
Public Class CaMarchePasException
```

```
    Inherits Exception
    Public Sub New()

    End Sub
    Public Sub New(ByVal message As String)
        MyBase.New(message)
    End Sub
    Public Sub New(ByVal message As String, ByVal inner As Exception)
        MyBase.New(message, inner)
    End Sub
End Class
```

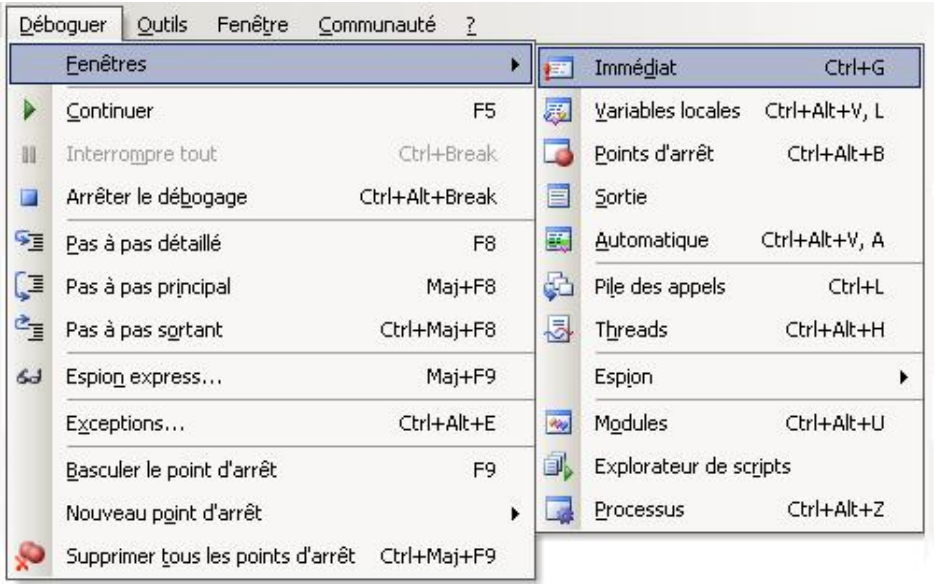
Cette classe peut ensuite être utilisée pour le déclenchement d'une exception personnalisée. Le code suivant déclenche une exception personnalisée dans un bloc `catch`.

```
    Catch ex As Exception
        Throw New CaMarchePasException("erreur dans l'application", ex)
    End Try
```

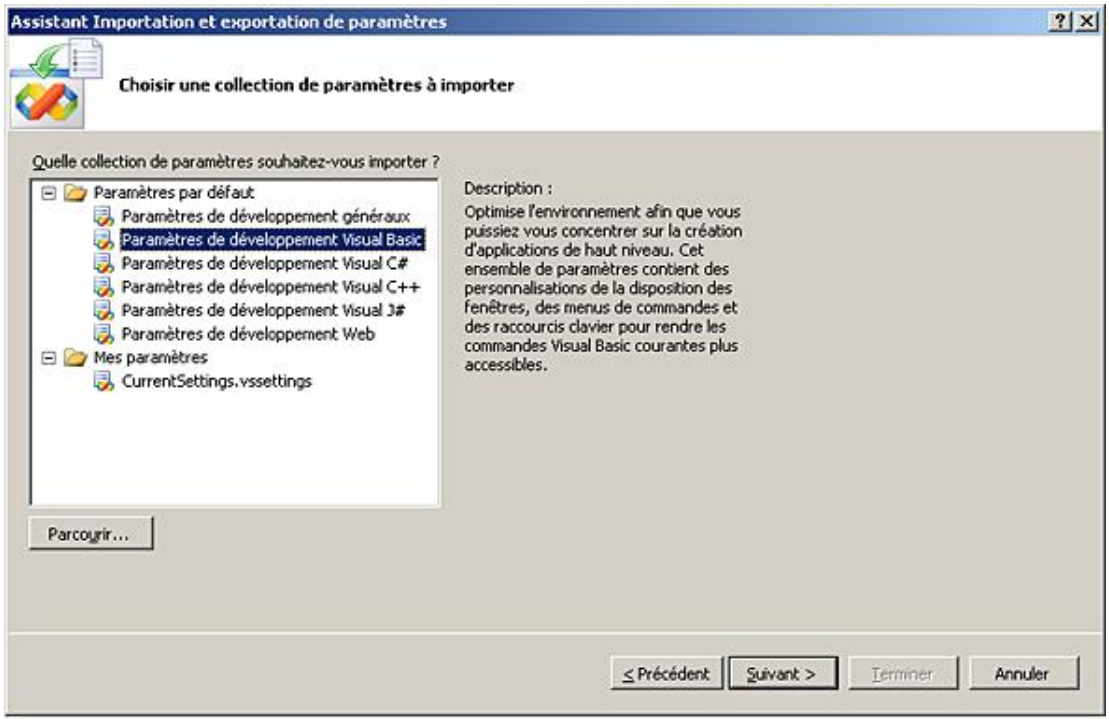
Les outils de débogage

Dans le chapitre consacré à la gestion des erreurs, nous avons vu que les erreurs de logique sont les plus difficiles à éliminer d'une application. Heureusement, Visual Studio.NET nous propose de nombreux outils de débogage à la fois performants et simples à utiliser. Ils permettent notamment de contrôler le déroulement de l'exécution de l'application (en plaçant des points d'arrêt et en faisant exécuter les instructions une par une), de visualiser et de modifier le contenu des variables, de visualiser le contenu de la mémoire à un emplacement particulier, de vérifier la liste de toutes les fonctions utilisées, etc. Ces différents outils sont accessibles par la barre d'outils **Déboguer**.

Le menu **Déboguer** fournit également l'accès à de nombreux outils :

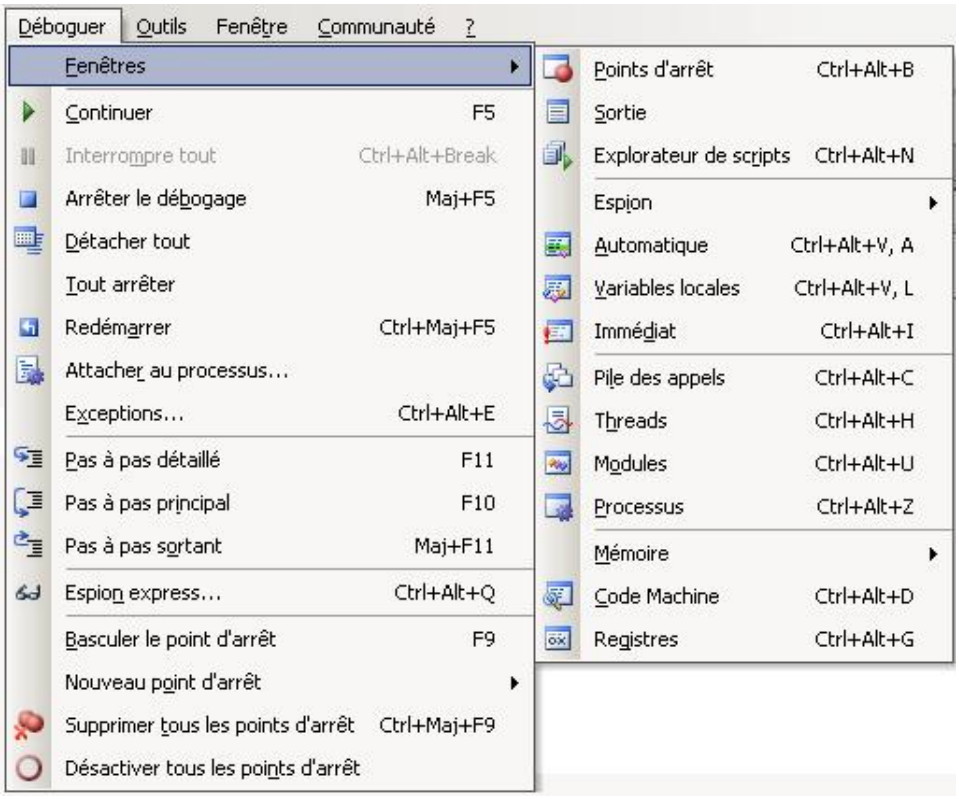


En fonction de la configuration de l'environnement de Visual Studio, certains outils ne seront peut-être pas disponibles. Vous pouvez reconfigurer Visual Studio pour intégrer ces outils supplémentaires par le menu **Outils - Importation et exportation de paramètres**. Les différentes boîtes de dialogue vous proposent de sauvegarder votre environnement actuel avant de le modifier, puis de choisir un environnement type à importer.



Parmi les configurations disponibles, c'est la configuration **Paramètres de développement généraux** qui propose le plus de fonctionnalités.

Après cette importation, de nombreux outils supplémentaires sont ajoutés au menu **Déboguer** :



Pour les explications suivantes de ce chapitre, nous allons considérer que c'est cette configuration qui est utilisée dans Visual Studio.

1. Contrôle de l'exécution

a. Démarrage de la solution

Un projet dans Visual Studio peut être dans trois états distincts :

- en conception
- en exécution
- en mode arrêt (l'exécution a été interrompue).

Le lancement de l'exécution peut s'effectuer par la barre d'outils ou par le raccourci-clavier [F5] ou [Ctrl] [F5]. Si, c'est cette dernière solution qui est utilisée, l'application est lancée en mode normal et aucun outil de débogage n'est disponible.

Si la solution contient plusieurs projets, l'un d'entre eux doit être configuré comme projet de démarrage pour la solution. Ce projet doit également avoir un objet de démarrage configuré, c'est par son exécution que va débuter l'application.

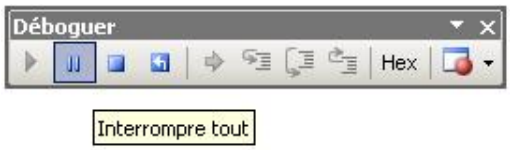
b. Arrêter la solution

L'arrêt de l'application peut s'effectuer en fermant toutes les fenêtres ; pour une application Windows, dès que la dernière fenêtre est fermée, l'application s'arrête ou par la combinaison de touches [Ctrl] **c** pour une application console. La barre d'outils ou le raccourci-clavier [Ctrl] [Alt] [Pause] permettent aussi d'arrêter l'application.

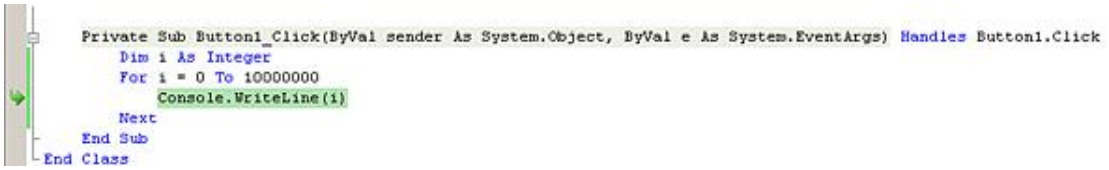
Le raccourci [Shift] [F5] permet de relancer l'exécution de la solution.

c. Interrompre la solution

L'interruption de l'exécution s'effectue avec la combinaison de touches [Ctrl] [Pause] ou par la barre d'outils :



L'interruption se produit sur l'instruction suivant celle en cours d'exécution au moment de la demande d'arrêt. La fenêtre de code devient à nouveau visible avec un repère en face de la ligne où l'exécution s'est interrompue.



Cette méthode n'est pas très pratique car il faut avoir beaucoup de chance pour interrompre l'exécution à un endroit précis. Nous verrons un peu plus loin que les points d'arrêts sont une bien meilleure solution pour interrompre l'exécution du code.

d. Poursuivre l'exécution

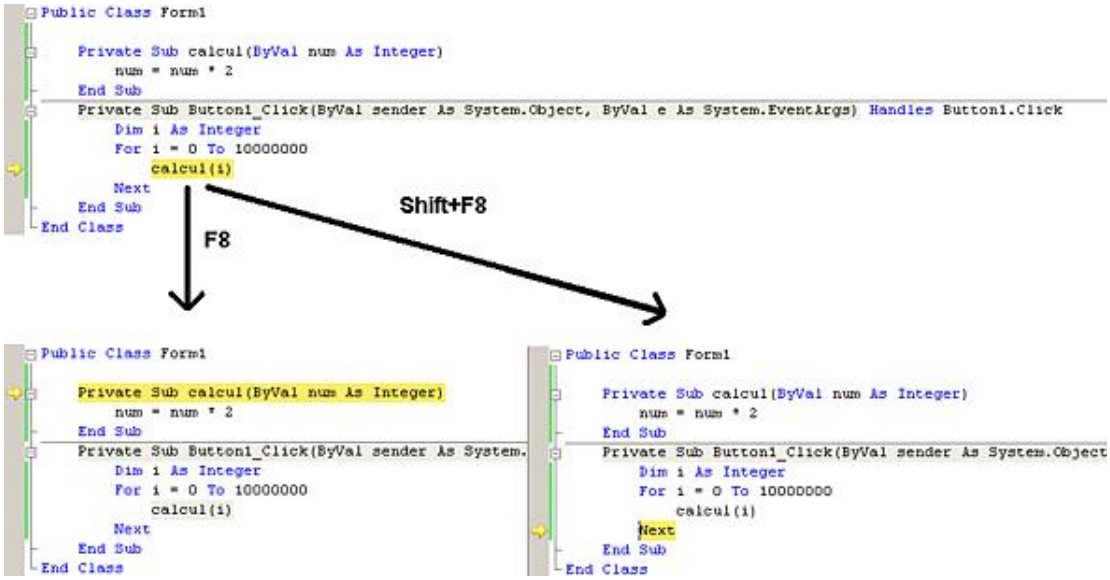
Une fois en mode arrêt, nous avons de nombreuses possibilités pour continuer l'exécution de l'application.

La première possibilité permet de reprendre l'exécution normale de l'application en utilisant la même technique que pour le démarrage du programme (barre d'outils ou raccourci-clavier [F5]). Cependant une technique plus courante lors d'un débogage est l'exécution en pas à pas.

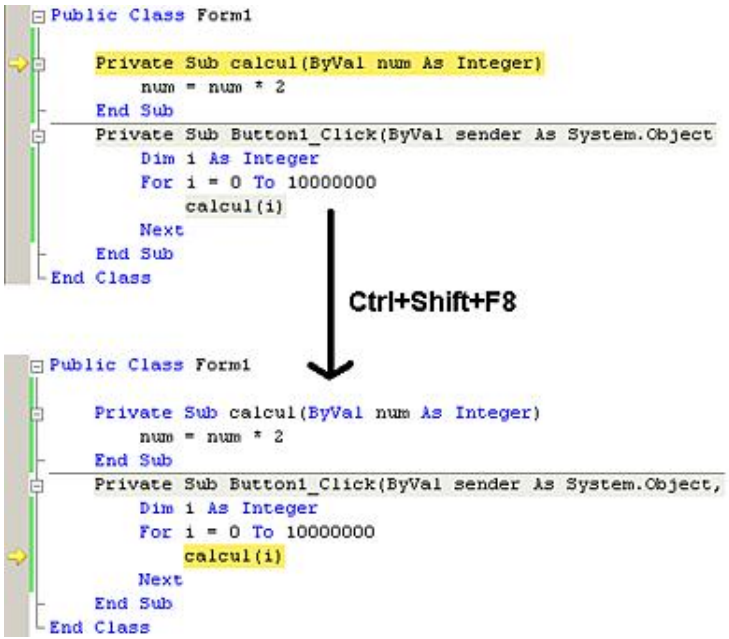
Trois solutions sont disponibles :

- Pas à pas détaillé ([F8])
- Pas à pas principal ([Shift] [F8])
- Pas à pas sortant ([Ctrl] [Shift] [F8])

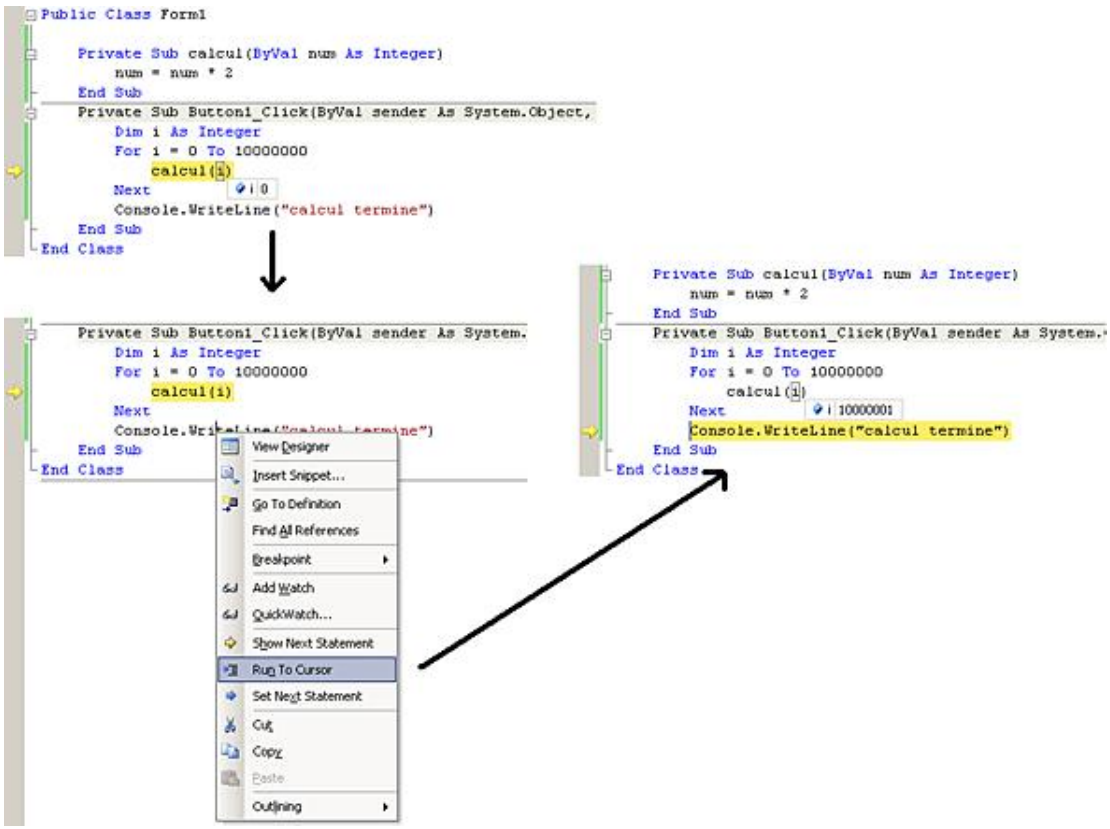
Le Pas à pas détaillé et le Pas à pas principal diffèrent simplement par leur façon de gérer les appels de procédures et fonctions. Si nous sommes en mode arrêt sur une ligne de code contenant un appel à une procédure ou une fonction, le mode Pas à pas détaillé va permettre de rentrer dans le code de la fonction puis de lancer l'exécution de son code ligne par ligne. Le mode Pas à pas principal exécutera la procédure ou la fonction en une seule fois sans que vous puissiez voir ce qui se passe à l'intérieur de la procédure ou fonction.



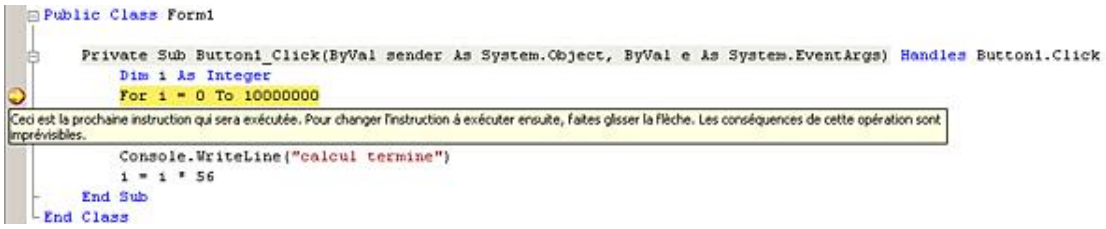
Le Pas à pas sortant permet l'exécution du code jusqu'à la fin d'une procédure ou fonction, sans décomposer ligne par ligne, puis repasse en mode arrêt sur la ligne suivant l'appel de la fonction.



Une dernière solution nous permet facilement d'exécuter un bloc de code puis de s'arrêter sur une ligne spécifique. Pour cela, un menu contextuel sur la fenêtre de code nous offre la possibilité de relancer l'exécution jusqu'à l'emplacement du curseur, sans s'arrêter sur toutes les instructions entre la ligne actuelle et la position du curseur (très utile pour exécuter rapidement toutes les itérations d'une boucle).



Inversement, si vous souhaitez ignorer l'exécution d'un bloc de code ou au contraire à nouveau exécuter un bloc de code, il est possible de déplacer le point d'exécution pour désigner la prochaine instruction exécutée. Il suffit de déplacer la flèche jaune affichée sur la marge en face de la prochaine instruction à exécuter.



Comme nous l'indique Microsoft, cette commande doit être utilisée avec précaution. Il faut notamment se souvenir des points suivants : les instructions situées entre l'ancien et le nouveau point d'exécution ne seront pas exécutées, déplacer le point d'exécution en arrière n'annule pas les instructions déjà traitées et le point d'exécution ne peut être déplacé qu'à l'intérieur d'une fonction ou procédure.

---

2. Points d'arrêt et TracePoint

Nous avons vu que la solution pour passer en mode arrêt était l'utilisation des touches [Ctrl] [Alt] [Pause]. Cette solution présente un gros inconvénient : l'exécution s'arrête n'importe où. Les points d'arrêt nous fournissent une solution plus élégante grâce à laquelle nous pouvons choisir l'emplacement où aura lieu l'interruption de l'exécution.

Les points d'arrêt peuvent aussi être conditionnels. Différents types de conditions sont pris en charge pour leur activation (condition, nombre de passage...).

Les TracePoint sont pratiquement identiques aux points d'arrêt mis à part que pour un TracePoint vous pouvez spécifier l'action exécutée lorsque le point est atteint. Ce peut être le passage en mode arrêt de l'application et/ou l'affichage d'un message. Les points d'arrêt ou les TracePoint sont affichés, dans l'environnement Visual Studio, par une série d'icônes. Les icônes vides représentent un élément désactivé.

 représente un point d'arrêt normal, activé ou désactivé.

 représente un point d'arrêt avancé (condition, nombre de passage ou filtre).


représente un TracePoint normal, activé ou désactivé.


représente un TracePoint avancé (condition, nombre de passage ou filtre).


représente un point d'arrêt ou un TracePoint en erreur.

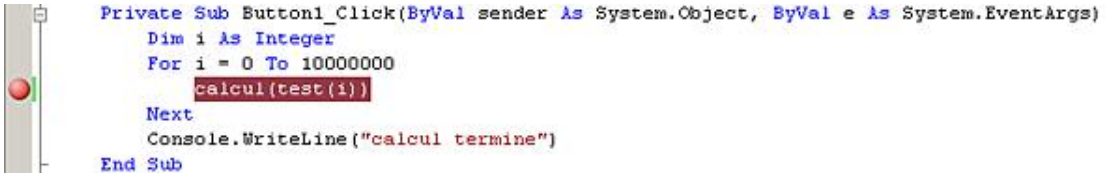

représente un avertissement sur un point d'arrêt ou un TracePoint.

a. Placer un point d'arrêt

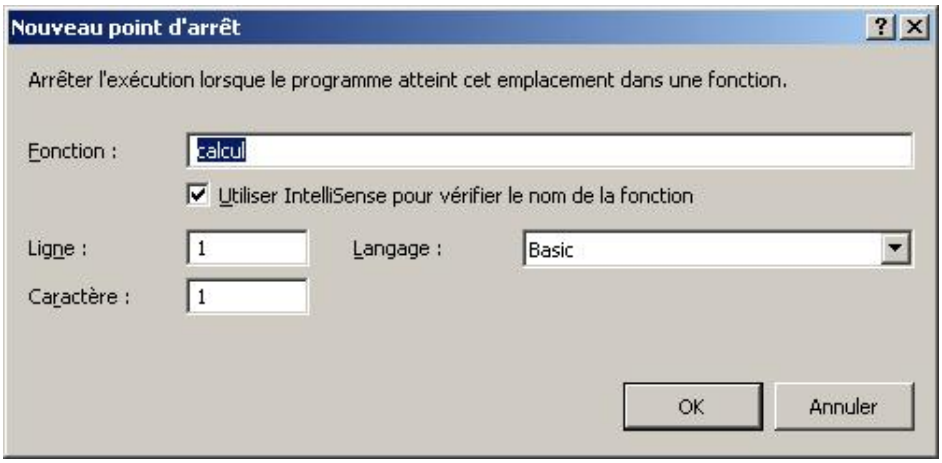
Pour placer un point d'arrêt, de nombreuses possibilités sont disponibles :

- effectuer un clic sur la marge de la fenêtre de code,
- positionner le curseur sur la ligne correspondante et utiliser le raccourci-clavier [Ctrl] **B**,
- utiliser l'option **Point d'arrêt - Insérer un point d'arrêt** du menu contextuel de la fenêtre de code.

Toutes ces techniques insèrent le point d'arrêt et matérialisent son emplacement par un point rouge dans la marge et le surlignement en rouge de la ligne correspondante.

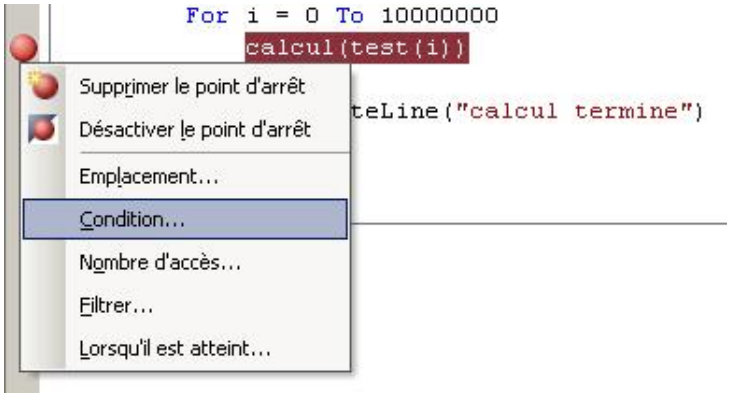


Pour toutes ces solutions, le code doit être visible dans l'éditeur. L'option **Interrompre à la fonction** du menu **Déboguer - Nouveau point d'arrêt** permet de placer un point d'arrêt sur une procédure ou fonction en saisissant simplement son nom.



Attention, la boîte de dialogue vous propose de préciser sur quelle ligne de la fonction vous souhaitez placer un point d'arrêt mais cette fonctionnalité n'est pas disponible pour les points d'arrêt sur des fonctions.

Les points d'arrêt ainsi placés sont inconditionnels. Dès que l'exécution arrive sur cette ligne, l'application passe en mode arrêt. On peut perfectionner le fonctionnement des points d'arrêt en y ajoutant des conditions, un nombre de passage ou en le transformant en TracePoint. Il convient pour cela de modifier les propriétés du point d'arrêt par le menu contextuel disponible par un clic droit sur la ligne concernée par le point d'arrêt.



Ajout d'une condition

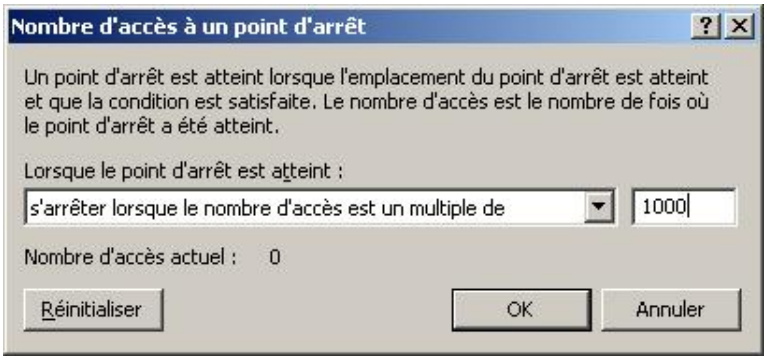
Le passage en mode arrêt peut être soumis à condition. La boîte de dialogue suivante permet de préciser les conditions d'exécution du point d'arrêt.

Nous devons saisir, dans cette boîte de dialogue, une expression qui sera évaluée à chaque passage sur le point d'arrêt. L'exécution s'arrêtera alors :

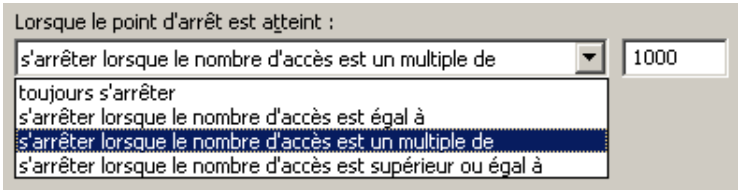
- si le résultat de l'évaluation de la condition est vraie,
- si le résultat de l'évaluation de la condition a été modifié depuis le dernier passage sur ce point d'arrêt. À noter que, dans ce cas, au moins deux passages sont nécessaires pour provoquer l'arrêt de l'application (le premier servant simplement à mémoriser le résultat de l'expression).

Modification du nombre de passages

Les points d'arrêt sont également capables de compter le nombre de fois où ils sont atteints et de s'exécuter pour un nombre particulier de passages.



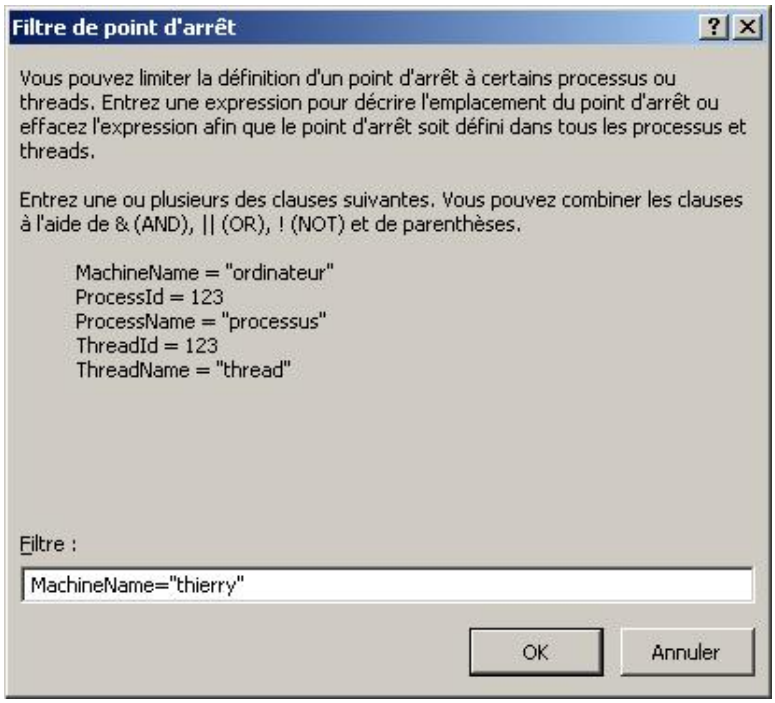
Cette boîte de dialogue nous permet de définir le nombre de passages sur le point d'arrêt pour que celui-ci arrête effectivement l'application. Quatre options sont disponibles pour la condition d'arrêt sur le nombre de passages.



Attention, si une condition est indiquée pour le point d'arrêt, le nombre de passage correspond au nombre de fois où l'exécution de l'application est passée sur cette ligne avec la condition vérifiée. Avec la configuration de notre exemple, nous nous arrêterons dans la boucle au 100000ième passage (la condition sera vraie pour i=0,100,200,300, 400,500,600,700,800,900).

Filtrage

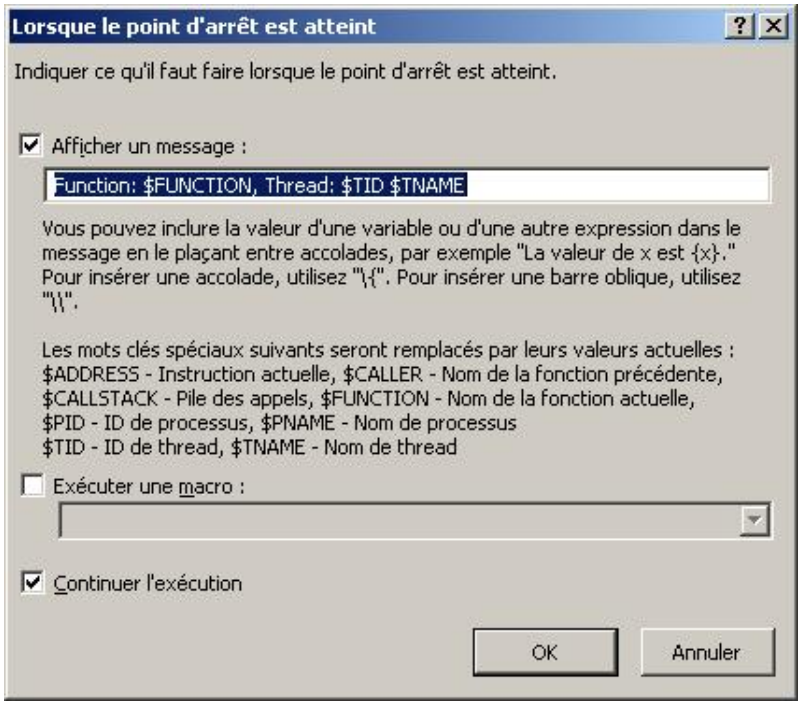
Les filtres permettent d'ajouter des critères supplémentaires pour l'exécution d'un point d'arrêt. Ces critères portent sur le nom de la machine où s'exécute l'application, ainsi que le processus ou le thread.



La condition doit être exprimée avec les mots clés **MachineName**, **ProcessId**, **ProcessName**, **ThreadId**, **ThreadName** et les opérateurs **&** (et), || **(ou)**, **! (not)**.

Transformation en TracePoint

Un point d'arrêt peut être transformé en TracePoint en précisant une action particulière à exécuter lorsqu'il sera atteint.



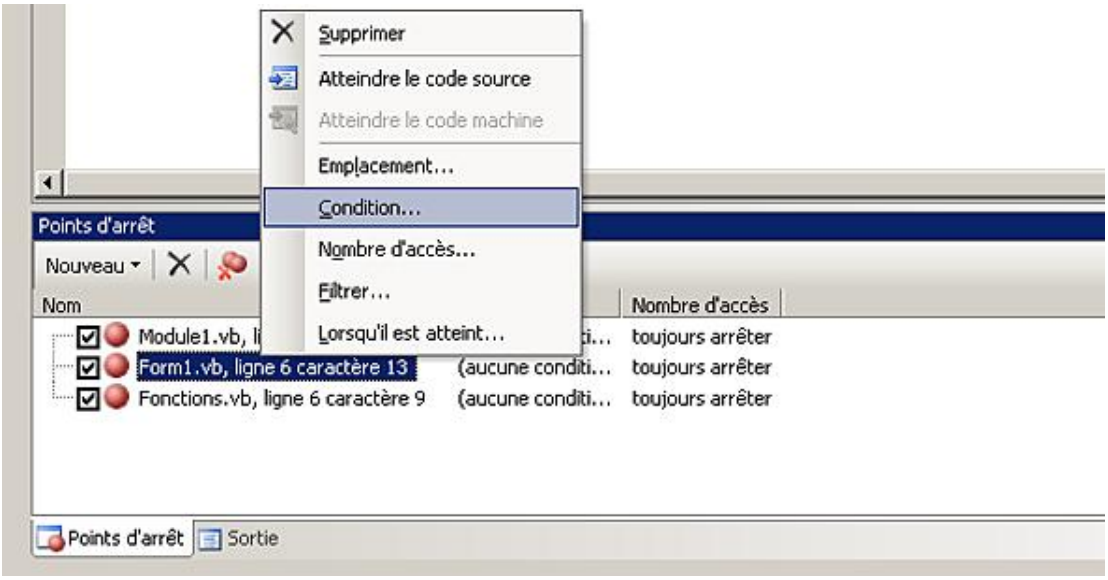
Cette boîte de dialogue attend le libellé du message affiché dans la fenêtre de sortie lorsque le point d'arrêt est atteint. Elle autorise également l'exécution d'une macro. Pour que le point d'arrêt soit vraiment transformé en TracePoint, l'option **Continuer l'exécution** doit être activée.

b. Activer, désactiver, supprimer un point d'arrêt

Les points d'arrêt peuvent également être momentanément désactivés en utilisant le menu contextuel.

Le point d'arrêt peut ensuite être à nouveau activé en utilisant à nouveau le menu contextuel. Ce même menu permet aussi la suppression d'un point d'arrêt, mais il est plus rapide d'effectuer un double clic sur le point d'arrêt lui-même. Le menu **Déboguer** propose également l'option **Supprimer tous les points d'arrêt**, évitant d'avoir à parcourir de nombreuses lignes de code pour éliminer l'ensemble des points d'arrêt.

Pour nous faciliter la tâche lors du débogage d'une application, une fenêtre nous propose un récapitulatif de tous les points d'arrêt placés dans votre projet. Cette fenêtre est accessible par l'intermédiaire du menu **Déboguer - Fenêtres - Points d'arrêt**. Cette fenêtre propose un menu contextuel permettant de réaliser les principales actions sur un point d'arrêt.



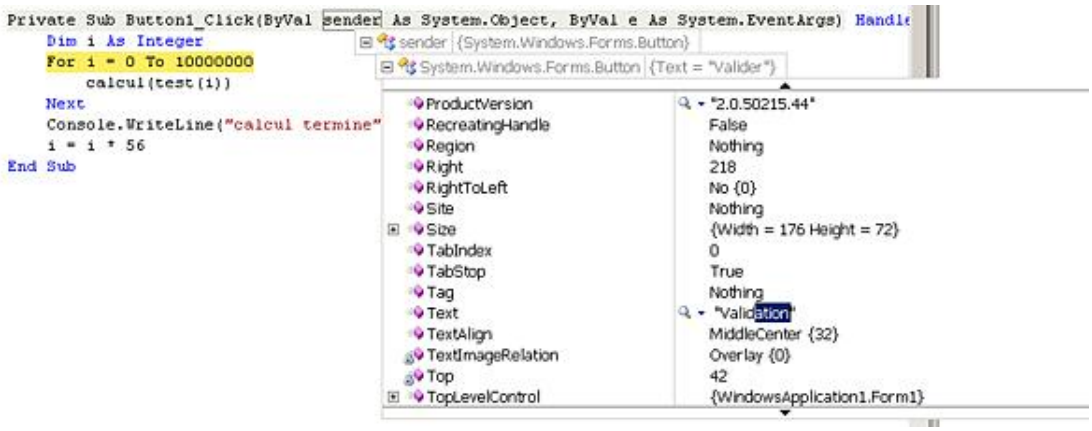
Contrairement à des versions plus anciennes de Visual Basic (version 6.0), les points d'arrêt sont conservés lorsque vous fermez votre projet.

3. Examen du contenu de variables

L'intérêt du débogueur est de pouvoir suivre le fonctionnement de l'application au cours de son fonctionnement. Il est également primordial de pouvoir visualiser, lorsque l'application est en mode arrêt, les valeurs contenues dans les différentes variables de l'application. Cette visualisation nous permet de vérifier le résultat des traitements déjà effectués ou d'anticiper sur les traitements effectués dans la suite du code.

a. DataTips

Les DataTips fournissent un moyen rapide pour visualiser le contenu d'une variable. Il suffit de déplacer le curseur de la souris au-dessus du nom et après, un court instant, une fenêtre présentant le contenu de la variable s'affiche. Si la variable est un type complexe, une instance de classe par exemple, le DataTips propose un petit signe + permettant de descendre dans la structure de la variable. Les informations visualisées sont également modifiables directement dans le DataTips. Le DataTips disparaît automatiquement lorsque la souris quitte sa surface.



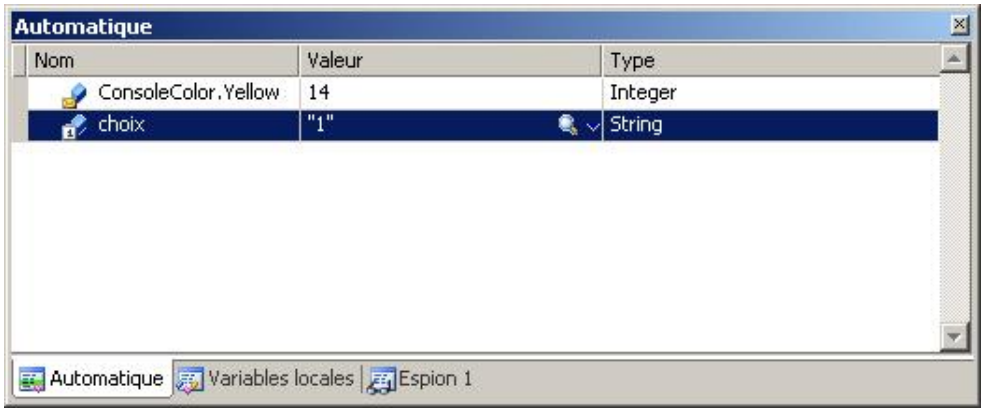
Pour afficher le résultat du calcul d'une expression, il convient au préalable de sélectionner l'expression puis de placer le curseur de la souris sur la sélection. Le débogueur évalue l'expression et affiche le résultat. Le DataTips ne peut afficher que les variables accessibles dans la portée courante (variables déclarées dans la fonction où l'on est arrêté ou variables globales).

Une petite astuce, si vous souhaitez visualiser le code masqué par le DataTips sans le faire disparaître, vous pouvez utiliser la touche [Ctrl] qui le rend transparent.

---

b. Fenêtre Automatique

La fenêtre **Automatique** affiche les variables utilisées dans l'instruction courante dans les trois instructions précédentes et dans les trois instructions suivantes. Cette fenêtre est accessible par le menu **Déboguer - Fenêtres - Automatique**.



Cette fenêtre permet également la modification du contenu d'une variable en double cliquant sur la valeur, dans la fenêtre, et en validant la modification, après saisie par la touche [Entrée]. L'application continuera à s'exécuter avec cette nouvelle valeur pour la variable.

c. Fenêtre Variables locales

La fenêtre **Variables locales** est accessible par le même **Déboguer - Fenêtres - Variables locales** et possède un fonctionnement identique à la fenêtre **Automatique** mis à part qu'elle affiche toutes les variables dans la portée actuelle, c'est-à-dire les variables globales et les variables déclarées dans la procédure ou fonction en cours d'exécution.

Dans toutes ces fenêtres, vous ne pouvez pas contrôler la liste des variables qui sont affichées puisque le débogueur en détermine la liste, en fonction du contexte dans lequel se trouve votre application. Il est parfois plus pratique de configurer manuellement la liste des variables et expressions à surveiller pendant le fonctionnement de l'application.

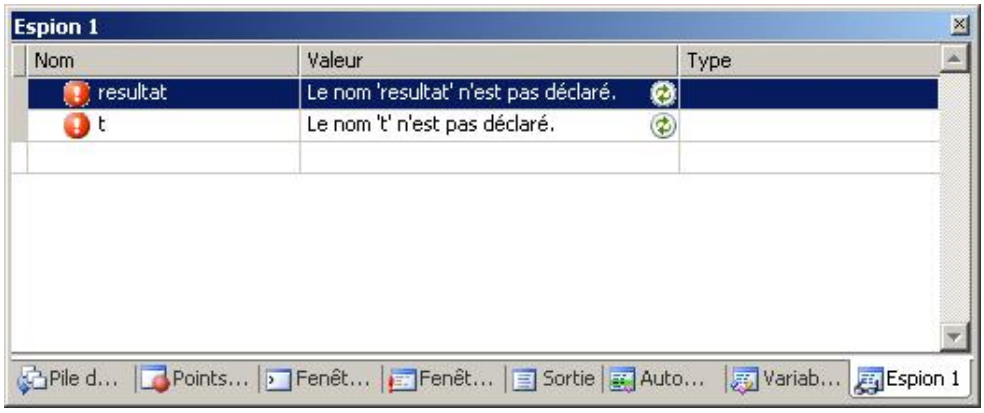
---

d. Les fenêtres Espion

La fenêtre **Espion** permet l'affichage des variables semblant intéressantes pour le débogage de l'application. Cette fenêtre ou plutôt ces fenêtres, puisque quatre fenêtres **Watch** sont disponibles, sont affichées par le menu **Déboguer - Fenêtres - Espion** puis **Espion 1** à **Espion 4**. Nous devons ensuite configurer la fenêtre en ajoutant les variables et expressions que nous souhaitons visualiser. En effectuant un double clic dans la colonne **Nom**, vous pouvez saisir ce que vous souhaitez afficher dans la fenêtre. Vous pouvez également effectuer un glisser-déplacer depuis la fenêtre de code. Si vous saisissez un nom de variable complexe (une instance de classe par exemple), l'ensemble de ses propriétés est affiché dans la fenêtre sous forme d'arborescence.

Le contenu des variables ne sera affiché que si l'application est en mode arrêt sur une ligne de code où cette variable peut être utilisée. Par exemple, le contenu des variables locales à une procédure ou fonction n'est affiché que si le code est arrêté dans cette procédure ou fonction.

Dans le cas contraire, la fenêtre **Espion** nous indique simplement que cette variable n'est pas déclarée dans la portion de code où nous nous trouvons, en l'affichant en caractères grisés.

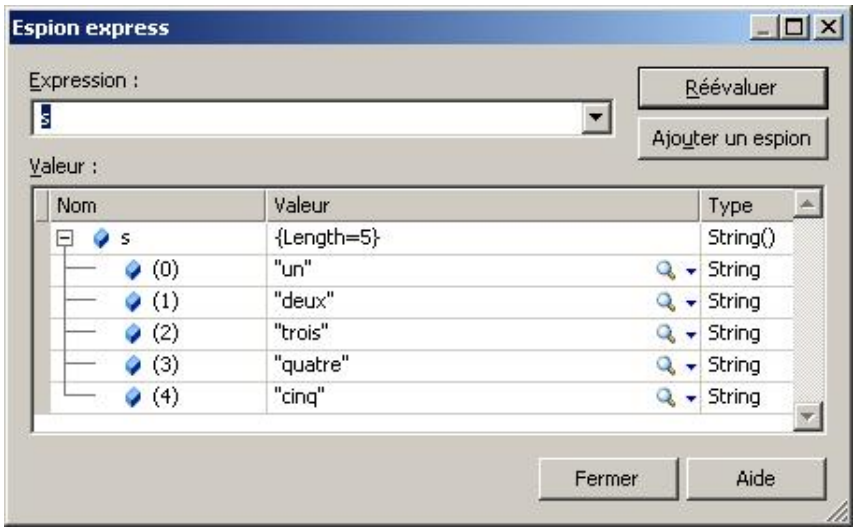


Dans cet exemple, les variables sont déclarées dans une autre procédure que celle où se trouve le pointeur d'exécution.

Comme pour les autres fenêtres, le contenu de la variable peut être modifié en double cliquant dessus pour passer en mode édition et en validant la saisie par la touche [Entrée].

e. La fenêtre Espion express

La fenêtre **Espion express** propose le même principe de fonctionnement et est accessible par le menu **Déboguer - Espion express**. Dans ce cas, la variable ou l'expression sur laquelle se trouve le curseur est affichée dans la fenêtre **Espion express**. Cette fenêtre étant modale, vous devrez obligatoirement la fermer avant de poursuivre le débogage de votre application.

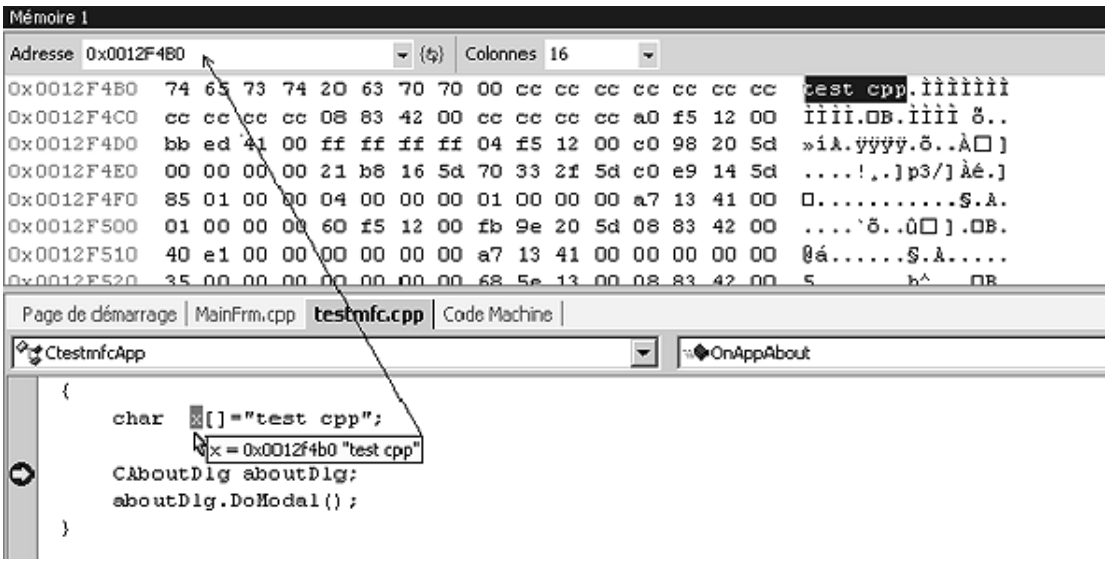


Le bouton **Ajouter un espion** permet d'ajouter rapidement l'expression dans la fenêtre **Espion** pour pouvoir l'étudier dans la suite du débogage.

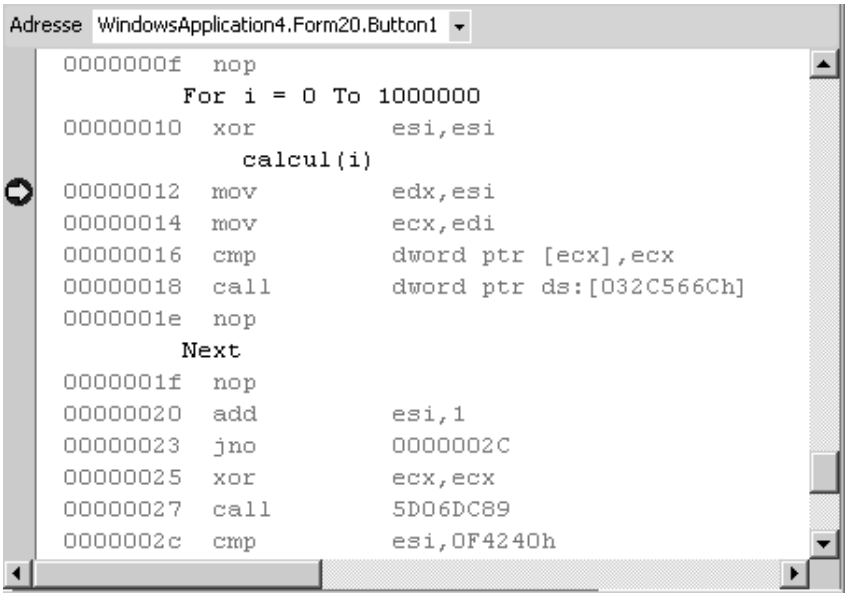
4. Les autres fenêtres de débogage

De nombreuses autres fenêtres sont disponibles pour le débogage mais certaines ne sont pas vraiment utiles pour le développement d'applications avec Visual Basic. Elles sont plutôt réservées pour le test d'applications développées avec d'autres langages, C++ par exemple.

Il s'agit, par exemple, de la fenêtre mémoire permettant la visualisation du contenu d'une zone mémoire dont on connaît l'adresse.



Si vous le désirez, vous pouvez visualiser le code machine correspondant aux instructions VB.NET.



Autres techniques de débogage

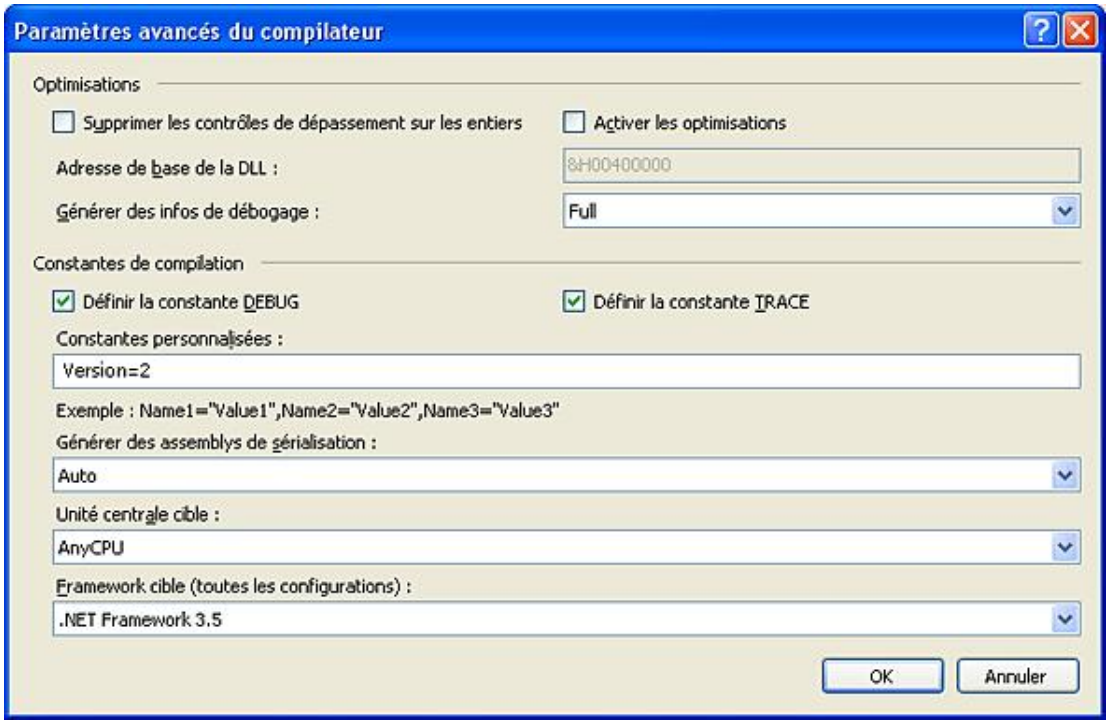
La compilation conditionnelle

Vous pouvez utiliser la compilation conditionnelle pour spécifier des portions de code qui seront ou non compilées, en fonction de la valeur d'une constante que vous aurez au préalable définie. Par exemple, vous pouvez tester plusieurs solutions pour résoudre un problème, en utilisant plusieurs algorithmes et vérifier le plus efficace d'entre eux.

Le bloc de code dont la compilation est soumise à condition doit être encadré par les instructions `#if condition Then` et `#endif` en fonction de la valeur de la condition, le bloc de code sera ou non compilé. Il faut bien sûr que la ou les variables utilisées dans la condition soi(en)t initialisée(s) avant son apparition dans une instruction `#if`.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles Button1.Click
        Dim i As Integer
#Const version = 2
#If version = 1 Then
        For i = 0 To 1000000
            calcul(i)
        Next
#End If
#If version = 2 Then
        Do While i < 100000
            calcul(i)
            i = i + 1
        Loop
#End If

End Sub
```

Les constantes peuvent être déclarées avec l'instruction `#const`, comme dans l'exemple ci-dessus, ou encore dans les propriétés du projet.



Attention cependant, car les constantes déclarées avec ces deux méthodes ne sont utilisables que pour la compilation conditionnelle et ne sont pas accessibles dans le code.

```
System.Console.WriteLine(Version)
```

'Version' est un type et ne peut pas être utilisé en tant qu'expression.

Utilisation de la classe Debug

La classe **Debug** dispose de plusieurs méthodes statiques, donc utilisables sans avoir à créer d'instance de la classe nous permettant d'insérer dans le code des instructions afin de suivre le déroulement de l'application.

Avec la méthode `assert`, vous affirmez que, lorsque cette ligne de code sera exécutée, la condition spécifiée dans la méthode sera vraie.

```
Debug.Assert(y <> 5)
```

Si la condition est fausse, Visual Basic affiche une boîte de dialogue résumant la situation.

Cette boîte de dialogue propose trois solutions pour continuer :

Abandonner

Arrête l'exécution de l'application.

Recommencer

L'application passe en mode arrêt sur la ligne contenant l'instruction `Assert`, permettant ainsi d'essayer d'analyser la situation.

Ignorer

L'exécution de l'application continue normalement.

La méthode `Fail` est plus radicale, puisqu'elle provoque l'affichage de la même boîte de dialogue, mais sans aucune condition. Elle peut être utilisée par exemple dans un bloc `Try End Try`.

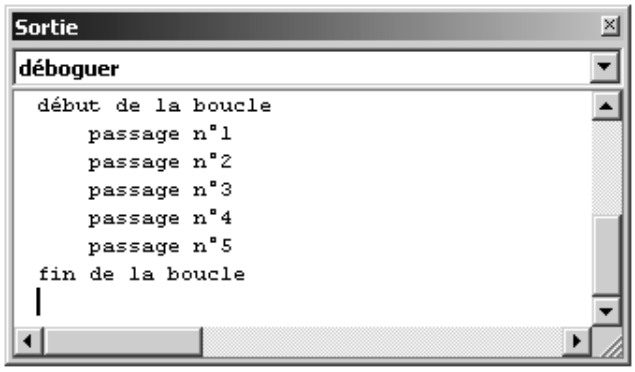
```
Try
    z = 1 / x
Catch ex As Exception
    Debug.Fail("division par zéro")
End Try
```

Les méthodes `Write` et `WriteLine` permettent de suivre l'évolution d'une expression sans que l'application passe en mode arrêt. L'instruction affichera, dans la fenêtre de sortie, le résultat de l'évaluation de l'expression.

Les méthodes `Indent` et `Unindent` formatent l'affichage dans la fenêtre **Sortie**.

```
Debug.WriteLine("début de la boucle")
Debug.Indent()
For x = 1 To 5
    Debug.WriteLine("passage n" & x)
    ...
    ...
Next
Debug.Unindent()
Debug.WriteLine("fin de la boucle")
```

Le résultat suivant est obtenu dans la fenêtre de sortie :



Les différents types d'application

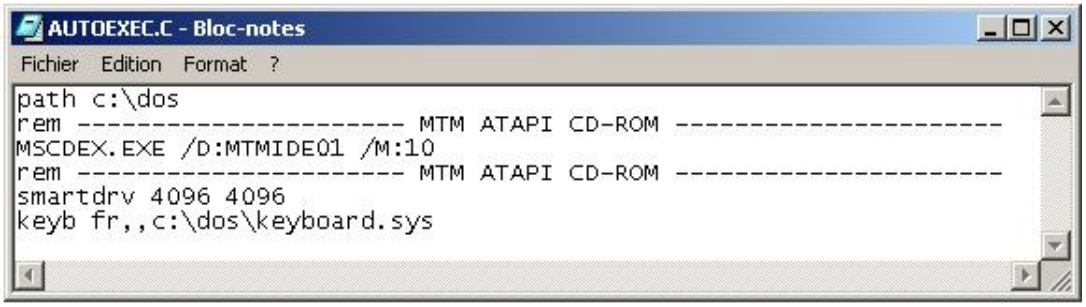
Les applications Windows sont basées sur une ou plusieurs fenêtres constituant l'interface entre l'utilisateur et l'application. Pour développer ce type d'application, nous avons à notre disposition dans le Framework.NET un ensemble de classes permettant la conception de l'interface de l'application. Ces éléments sont fréquemment regroupés sous le terme Technologie Windows Forms. Une application basée sur les Windows Forms utilisera un ou plusieurs formulaires pour construire l'interface utilisateur de l'application. Sur ces formulaires (ou feuilles), nous placerons des contrôles afin de définir exactement l'aspect de l'interface de l'application. Les formulaires seront créés à partir de classes du Framework.NET que l'on spécialisera par l'ajout de fonctionnalités. Le formulaire ainsi créé étant lui-même une classe, il sera possible de le réutiliser, dans une autre application, en lui ajoutant des fonctionnalités supplémentaires par une relation d'héritage. Les Windows Forms peuvent être créés directement par le code mais l'environnement de développement Visual Studio propose toute une panoplie d'outils graphiques pour nous faciliter la tâche. Nous utiliserons principalement cette technique.

1. Modes de présentation des fenêtres

Dans une application Windows, trois styles de présentation sont disponibles pour les fenêtres de l'application.

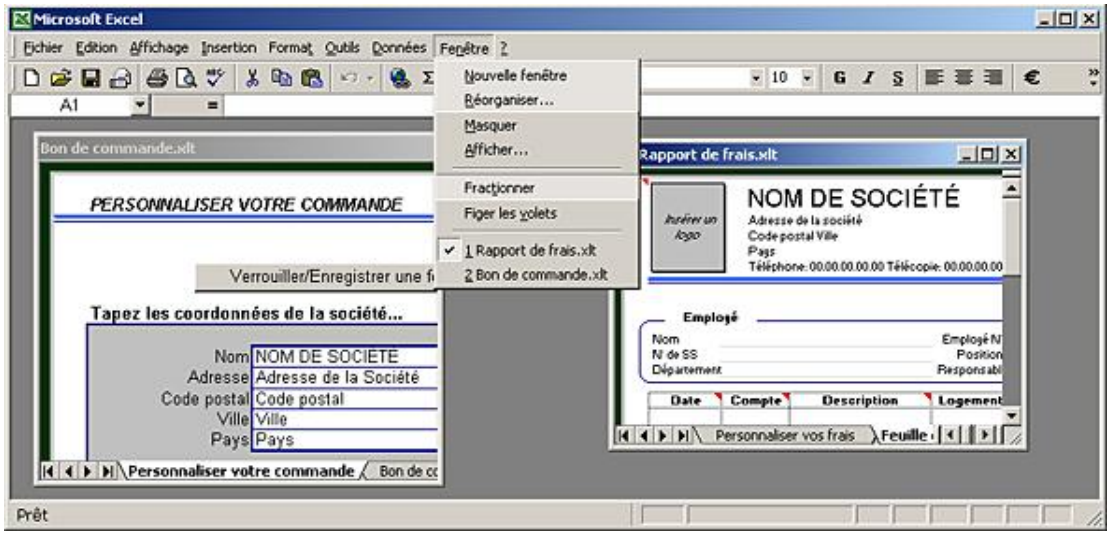
a. Interface mono document (SDI)

Une seule fenêtre est disponible dans l'application. Pour pouvoir ouvrir un nouveau document, le document actif de l'application doit obligatoirement être fermé. Le bloc-notes de Windows est une application SDI.

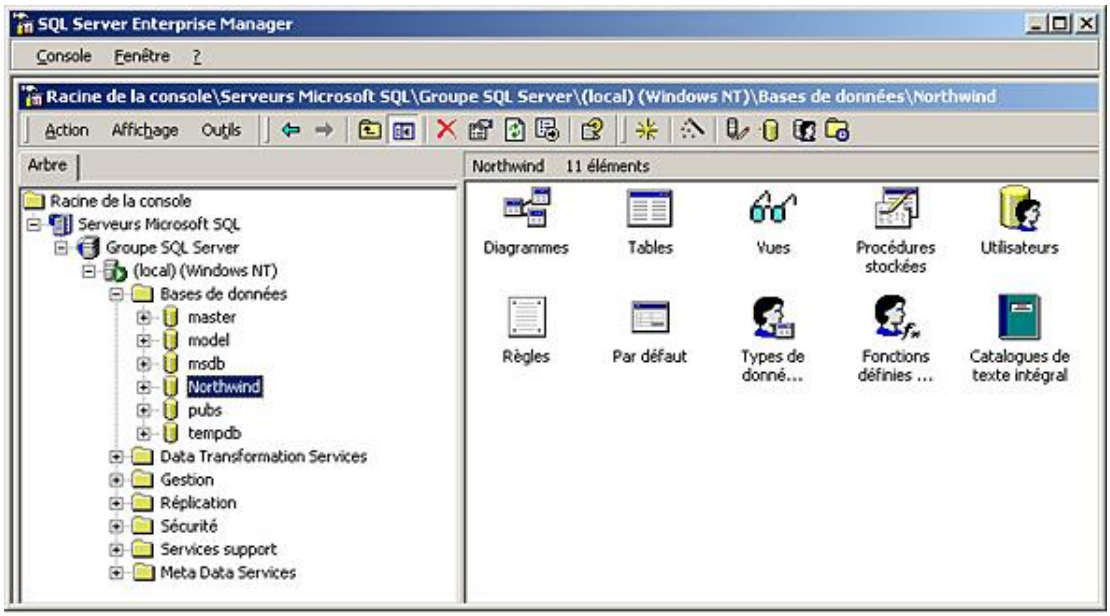


b. Interface multidocuments (MDI)

L'application est constituée d'une fenêtre principale (la fenêtre mère) dans laquelle vont apparaître plusieurs fenêtres (fenêtres filles) contenant les documents sur lesquels vous souhaitez travailler. En règle générale, ce type d'application dispose d'un menu permettant la réorganisation des différentes fenêtres filles. Ce type de présentation est utilisé par la plupart des applications bureautiques.



c. Interface de style explorateur



Ce style d'interface se développe de plus en plus au détriment des deux autres. Dans ce cas, la fenêtre est divisée en deux zones. La zone de gauche présente une vue, sous forme d'arborescence, des éléments pouvant être manipulés par l'application. La zone de droite présente l'élément sélectionné dans l'arborescence et permet sa modification. De nombreux outils d'administration utilisent cette présentation.

Les fenêtres dans VB.NET

Lorsque vous commencez une nouvelle application Windows Forms, l'environnement de développement ajoute automatiquement au projet un formulaire. Ce formulaire sert de point de départ pour l'application. Vous pouvez immédiatement lancer l'exécution de la solution et tout fonctionne. Certes, l'application ne permet pas d'effectuer grand-chose, mais elle a toutes les fonctionnalités d'une application Windows et, cela, sans écrire une seule ligne de code. En fait, il existe bien du code correspondant à cette application mais il a été généré automatiquement par Visual Studio. Comme ce code ne doit jamais être modifié manuellement, les fichiers le contenant sont masqués dans l'explorateur de solutions. Pour les afficher, vous pouvez utiliser le bouton  de la barre d'outils de l'explorateur de solutions. Vous pouvez alors constater que de nombreux fichiers existent déjà dans le projet. Les fichiers réservés de Visual Studio ont tous l'extension .designer.vb. Vous pouvez bien sûr visualiser le contenu de ces fichiers.

Voici par exemple le contenu du fichier Application.designer.vb.

```
'------------------------------------------------
' <auto-generated>
'     This code was generated by a tool.
'     Runtime Version:2.0.50727.1433
'
'     Changes to this file may cause incorrect behavior and will be lost if
'     the code is regenerated.
' </auto-generated>
'------------------------------------------------

Option Strict On
Option Explicit On


Namespace My

    'NOTE: This file is auto-generated; do not modify it directly. To make changes,
    ' or if you encounter build errors in this file, go to the Project Designer
    ' (go to Project Properties or double-click the My Project node in
    ' Solution Explorer), and make changes on the Application tab.
    '
    Partial Friend Class MyApplication

        <Global.System.Diagnostics.DebuggerStepThroughAttribute()>  _
        Public Sub New()

MyBase.New(Global.Microsoft.VisualBasic.ApplicationServices.AuthenticationMode.Windows)
            Me.IsSingleInstance = false
            Me.EnableVisualStyles = true
            Me.SaveMySettingsOnExit = true
            Me.ShutDownStyle =
Global.Microsoft.VisualBasic.ApplicationServices.ShutdownMode.AfterMainFormCloses
        End Sub

        <Global.System.Diagnostics.DebuggerStepThroughAttribute()>  _
        Protected Overrides Sub OnCreateMainForm()
            Me.MainForm = Global.WindowsApplication1.Form1
        End Sub
    End Class
End Namespace
```

Les commentaires placés dans ce fichier sont suffisamment clairs : ne jamais modifier ce fichier manuellement ! Il est mis à jour automatiquement à chaque modification des propriétés du projet. Le deuxième fichier, mis à jour automatiquement, est associé à la fenêtre de l'application. Dans notre première application, il est nommé form1.designer.vb. Il va contenir la description de toutes les actions, traduites en code VB, que vous allez réaliser pour personnaliser les caractéristiques de la fenêtre.

Regardons le contenu de ce fichier :

```
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class Form1
    Inherits System.Windows.Forms.Form

    'Form overrides dispose to clean up the component list.
```

```
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        Try
            If disposing AndAlso components IsNot Nothing Then
                components.Dispose()
            End If
        Finally
            MyBase.Dispose(disposing)
        End Try
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.

    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        components = New System.ComponentModel.Container()
        Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
        Me.Text = "Form1"
    End Sub
```

Il contient la définition de la classe correspondant à la fenêtre héritant de la classe `System.Windows.Forms.Form`. Cette définition comporte une petite particularité par rapport à la définition d'une classe telle que nous l'avons vue dans le chapitre consacré à la programmation objet. Le mot clé `Partial` est spécifié devant le nom de la classe. Ce mot clé indique au compilateur que le fichier ne contient qu'une partie de la définition de la classe, l'autre partie étant disponible dans le fichier Form1.vb. Cette technique permet de répartir les rôles :

- Visual Studio se charge de générer, dans le fichier form1.designer.vb, le code correspondant à la personnalisation de l'aspect de la fenêtre.
- Vous êtes responsable du code, contenu dans le fichier form1.vb, chargé de la personnalisation du fonctionnement de la fenêtre.

Cette solution limite les risques de modification involontaire de la partie de code réservée à Visual Studio. L'élément le plus important est constitué par la méthode `InitializeComponent`. Cette méthode est appelée automatiquement à la création d'une instance de la fenêtre, lors de l'appel du constructeur. Pour vous en convaincre, vous pouvez ajouter un constructeur par défaut dans le fichier Form1.vb et constater que Visual Studio le transforme automatiquement en y ajoutant un appel à cette méthode.

```
Public Class Form1
    Public Sub New()
        ' This call is required by the Windows Form Designer.
        InitializeComponent()
        ' Add any initialization after the InitializeComponent() call.
    End Sub
```

Par contre, si vous ajoutez un constructeur surchargé, vous êtes responsable de cet appel. Le plus simple dans ce cas étant de faire appel au constructeur, par défaut, à la première ligne de votre constructeur surchargé.

```
    Public Sub New(ByVal i As Integer)
        MyClass.New()
    End Sub
```

Pensez également à bien respecter, dans le constructeur par défaut, l'emplacement qui vous est réservé pour vos initialisations particulières. Si elles sont placées avant l'appel à la méthode `InitializeComponent`, celui-ci risque de les modifier. Avant l'appel de la méthode `InitializeComponent` les éléments graphiques de la fenêtre ne sont pas disponibles car c'est le rôle principal de cette méthode de les créer et d'initialiser certaines de leurs propriétés.

Une méthode `dispose` est également créée pour pouvoir supprimer tous les objets instanciés par la classe. Cette méthode commence par supprimer les objets créés puis elle appelle la méthode `dispose` de la classe parente.

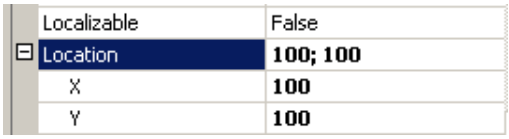
```
    'Form overrides dispose to clean up the component list.
    <System.Diagnostics.DebuggerNonUserCode()> _
```

```
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing AndAlso components IsNot Nothing Then
            components.Dispose()
        End If
        MyBase.Dispose(disposing)
    End Sub
```

Nous avons fait le tour du code généré automatiquement. Regardons maintenant comment modifier l'apparence et le comportement de notre fenêtre par l'intermédiaire de ces propriétés.

1. Dimension et position des fenêtres

La position de la feuille sur l'écran (ou sur son conteneur) est modifiée par la propriété `location`. Cette propriété est une structure composée de deux membres, indiquant les coordonnées du coin supérieur gauche de la feuille par rapport au coin supérieur gauche de l'écran. Les membres de cette structure peuvent être modifiés dans la fenêtre de propriétés de vb. En fait, lorsque vous modifiez des propriétés, du code est ajouté pour prendre en compte vos modifications. Par exemple, si nous voulons que notre fenêtre apparaisse aux coordonnées 100 100, nous modifions la propriété `Location`.

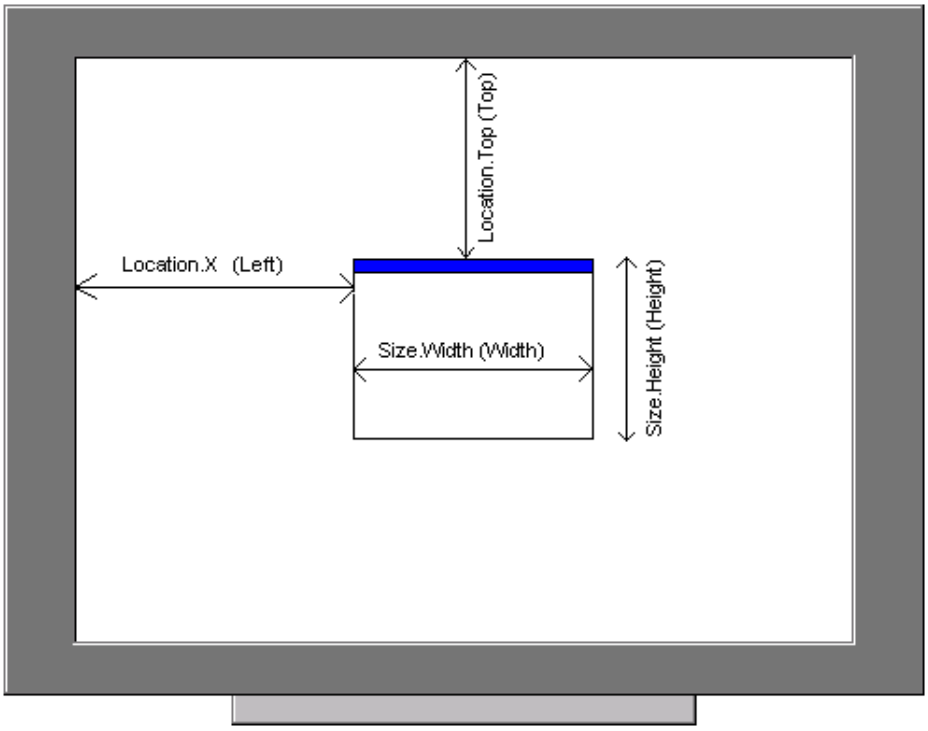


Si nous regardons dans le code, nous retrouvons la modification de notre propriété.

```
Me.Location = New System.Drawing.Point(100, 100)
```

Les dimensions de la fenêtre sont modifiables par la propriété `size` qui contient deux membres `Width` et `Height` indiquant la largeur et la hauteur de la fenêtre.

Résumons tout cela par un petit schéma :



Les unités sont des pixels, pour toutes les propriétés, concernant les dimensions et positions d'objet. Les propriétés `Left`, `Top`, `Height` et `Width` sont disponibles dans le code mais pas dans la fenêtre de propriétés. La correspondance avec les propriétés `Location` et `Size` de ces propriétés est indiquée entre parenthèses sur le schéma.

Ces propriétés sont mises à jour pendant l'exécution de l'application, si la fenêtre est déplacée ou redimensionnée. Elles sont accessibles par le code de l'application.

La largeur et la hauteur de la fenêtre peuvent évoluer entre les limites fixées par les propriétés `MinimumSize` et

MaximumSize. Par défaut ces deux propriétés sont initialisées à 0 ; 0 indiquant, dans ce cas, qu'il n'y a pas de limite fixée pour la taille de la fenêtre.

Deux autres propriétés indiquent le comportement de la fenêtre au démarrage de l'application.

La propriété StartPosition permet d'imposer une position à la fenêtre, au démarrage de l'application. Les valeurs possibles sont résumées dans le tableau suivant :

Valeur de la propriété	Effet sur la fenêtre
Manual	Les propriétés Location et Size sont utilisées pour l'affichage de la fenêtre.
CenterParent	La fenêtre est centrée dans la fenêtre mère.
CenterScreen	La fenêtre est centrée sur l'écran.
WindowsDefaultLocation	Le système positionne automatiquement les fenêtres en partant du coin supérieur gauche de l'écran. Les fenêtres sont décalées vers le bas droit de l'écran à chaque affichage d'une nouvelle fenêtre. La taille de chaque fenêtre est spécifiée par la propriété Size.
WindowsDefaultBounds	Même principe que ci-dessus, mais la taille est déterminée par le système à l'affichage de la fenêtre.

La propriété WindowState indique l'état de la feuille. Les trois valeurs possibles sont :

Valeur de la propriété	État de la fenêtre
Normal	Taille définie par la propriété Size.
Minimized	Fenêtre en icône sur la barre des tâches.
Maximized	Fenêtre en plein écran.

Ces propriétés peuvent, bien sûr, être modifiées par le code de l'application. Par contre, il est plus efficace d'utiliser les méthodes SetLocation et SetSize qui permettent directement le dimensionnement et le positionnement de la feuille. L'utilisation de ces méthodes ou la manipulation directe des propriétés déclenchent les événements Resize et Move sur la feuille correspondante.

Le code suivant nous permet de suivre la position et la dimension de la feuille :

```
Private Sub Form1_Resize(ByVal sender As Object, ByVal e As System.Event
Args)Handles MyBase.Resize

System.Console.WriteLine(" Ma largeur : " & Me.Size.Width)
System.Console.WriteLine(" Ma hauteur : " & Me.Size.Height)
End Sub
Private Sub Form1_Move(ByVal sender As Object, ByVal e As System.EventArgs)
   Handles MyBase.Move
System.Console.WriteLine(" Je suis à la position X :" & Me.Location.X )
System.Console.Writeline(" Je suis à la position Y :" & Me.Location.Y)
End Sub
```

Nous obtenons les informations suivantes :

```
Je suis à la position X :263
Je suis à la position Y :311
Ma largeur : 364
Ma hauteur : 122
```

Une petite curiosité pour terminer avec la taille et position des fenêtres. Si nous réduisons notre fenêtre en icône, en cliquant sur le bouton **Réduire**  ou en modifiant la propriété WindowState, nous obtenons les valeurs suivantes :

```
Je suis à la position X :-32000
Je suis à la position Y :-32000
Ma largeur : 160
```

```
Ma hauteur : 24
```

Les positions X et Y de la feuille sont bien des valeurs négatives ! En fait, vous pouvez utiliser des valeurs comprises dans la limite de celles acceptables pour un entier. Seule la partie de votre fenêtre comprise entre zéro et la largeur et hauteur de votre écran sera visible. Vous pouvez d'ailleurs utiliser la méthode `GetBounds` de l'objet `Screen` pour obtenir la taille de l'écran.

```
    Private Sub Form1_Move(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Move

    System.Console.WriteLine(" Je suis à la position X :" & Me.Location.X)
    System.Console.WriteLine(" Je suis à la position Y :" & Me.Location.Y)
    System.Console.Write(" sur un écran de :" & Screen.GetBounds(Me).Size.Width)
    System.Console.WriteLine(" par " & Screen.GetBounds(Me).Size.Height)
   End Sub
```

Ce code nous permet de connaître la dimension de l'écran qui affiche l'application.

```
Je suis à la position X : 115
Je suis à la position Y : 203
sur un écran de : 1024 par 768
```

Pour que l'utilisateur puisse déplacer ou modifier la taille de la fenêtre, il doit disposer des outils nécessaires :

- une barre de titre pour pouvoir agripper la fenêtre et la déplacer ;
- une bordure pour pouvoir la dimensionner ;
- des boutons pour pouvoir l'agrandir, la réduire et l'afficher avec sa taille normale.

Pour pouvoir redimensionner la fenêtre, celle ci doit disposer d'une bordure de type "sizable" affectée à sa propriété `FormBorderStyle`.

Pour être déplacée, une fenêtre doit avoir une barre de titre. Cette barre de titre peut être masquée par la propriété `ControlBox` positionnée sur `False` ; dans ce cas, même le titre de la fenêtre spécifié par la propriété `Text` n'est plus affiché. Dans le cas où la barre de titre est affichée, l'état des différents boutons apparaissant dessus peut être contrôlé avec les propriétés suivantes :

`MinimizeBox`

Affichage ou non du bouton de mise en icône de la fenêtre.

`MaximizeBox`

Affichage ou non du bouton d'agrandissement de la fenêtre.

`HelpButton`

Affichage du bouton d'aide. Visible seulement si les deux boutons précédents ne sont pas visibles.

2. Couleurs et Police utilisées sur les fenêtres

La propriété `BackColor` indique la couleur de fond utilisée sur la fenêtre. Cette couleur sera également utilisée pour tous les contrôles qui seront, par la suite, placés sur la feuille. La propriété `ForeColor` indique la couleur des éléments qui seront tracés directement sur la feuille ou la couleur de la légende des contrôles placés sur cette feuille. Il existe quatre possibilités pour affecter une valeur à ces propriétés de couleur :

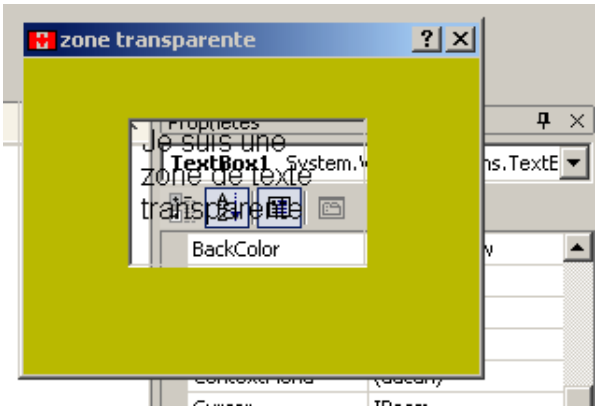
- par la fenêtre de propriétés, en choisissant une couleur dans l'onglet **Personnaliser**.
- par la fenêtre de propriétés, en choisissant une couleur web. Ces couleurs correspondent aux couleurs disponibles en langage HTML.
- par la fenêtre de propriétés, en choisissant une couleur système. Dans ce cas, votre application s'adaptera automatiquement à l'environnement du poste de travail sur lequel elle est installée. Si l'utilisateur a configuré son poste pour avoir des boutons de couleur rose fluo, il retrouvera la même apparence dans votre application.

- En fabriquant vous-même votre couleur avec un peu de rouge, un peu de vert, un peu de bleu. Pour mélanger tout cela et obtenir la couleur finale, utilisez la méthode`FromARGB` qui prend comme paramètre la quantité de rouge, la quantité de vert, la quantité de bleu et fournit la couleur résultante. Les quantités de chaque couleur sont des valeurs comprises entre 0 et 255, cette dernière valeur correspondant à une couleur pure.

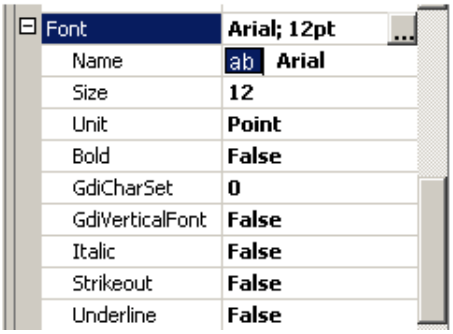
La propriété `Opacity` permet de régler la transparence de votre feuille. La valeur doit être comprise entre zéro (fenêtre transparente) et un (fenêtre opaque). Attention, cette propriété ne sera prise en compte que sur certains systèmes (Windows 2000, Windows XP, Windows 2003, Vista). Vous pouvez également faire de la tapisserie en indiquant une image de fond pour votre fenêtre avec la propriété `BackgroundImage`. Si l'image n'est pas assez grande pour couvrir la fenêtre, elle est reproduite en mosaïque.

De la même manière, vous pouvez spécifier qu'une couleur sera considérée transparente sur votre fenêtre. Pour cela, vous devez affecter à la propriété `TransparencyKey` la valeur de cette couleur.

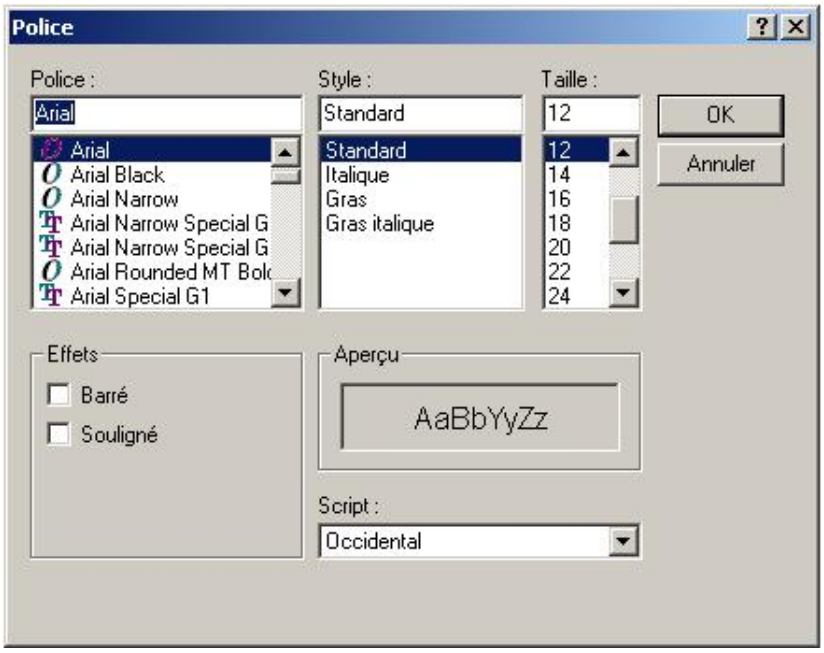
Pour illustrer l'utilisation peu évidente de cette propriété, nous avons indiqué dans la fenêtre ci-dessous que la couleur blanche était transparente (on aperçoit une partie de la fenêtre de propriétés à travers la zone de texte).



La propriété `Font` permet de spécifier les caractéristiques de la police de caractères, utilisée pour l'affichage de texte, directement sur la fenêtre. Cette police sera également utilisée pour tous les contrôles que l'on placera sur la fenêtre. Vous pouvez modifier les propriétés directement dans la fenêtre de propriété, en déroulant la propriété `Font` en cliquant sur le signe plus (+) en face de la propriété.



Vous pouvez également modifier les caractéristiques de la police avec la boîte de dialogue standard de choix de police. Celle-ci est affichée en utilisant le bouton ... , en regard de la propriété `Font` dans la fenêtre de propriétés.



3. Les fenêtres MDI

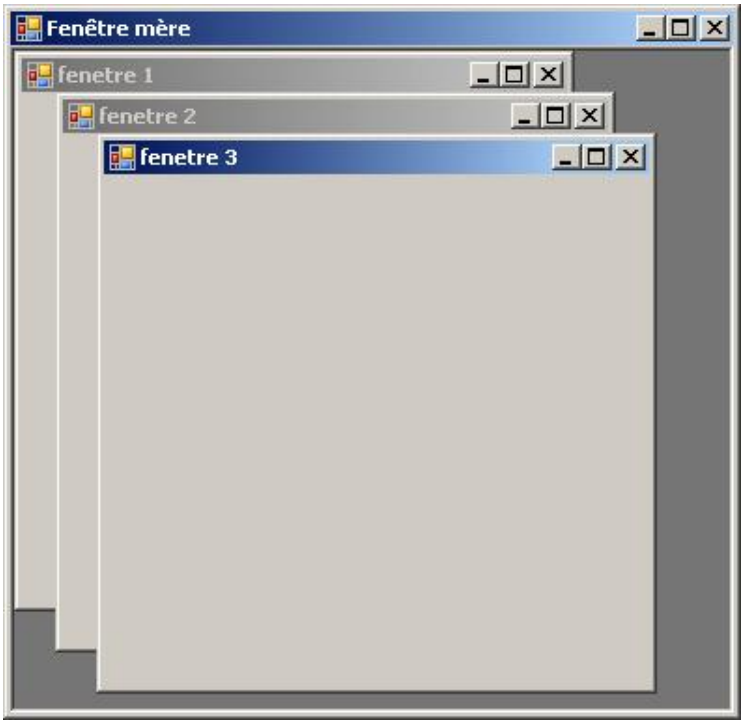
Les applications MDI sont constituées de deux types de feuilles :

- les feuilles mères,
- les feuilles filles.

Dans VB.NET, la même classe de base est utilisée pour les deux types de fenêtre. Dans le premier cas, on indiquera simplement le fait que la fenêtre est une fenêtre mère MDI en positionnant sur **True** sa propriété `IsMdiContainer`. Pour ajouter ensuite une fenêtre fille, il convient bien sûr d'abord de créer la fenêtre puis de l'associer à une fenêtre mère par sa propriété `MdiParent`.

Voici un code qui crée trois fenêtres et les transforme en fenêtres filles MDI :

```
        Dim fenetre1 As Form
        Dim fenetre2 As Form
        Dim fenetre3 As Form

        fenetre1 = New Form()
        fenetre1.Text = "fenetre 1"
        fenetre1.MdiParent = Me
        fenetre1.Show()
        fenetre2 = New Form()
        fenetre2.Text = "fenetre 2"
        fenetre2.MdiParent = Me
        fenetre2.Show()
        fenetre3 = New Form()
        fenetre3.Text = "fenetre 3"
        fenetre3.MdiParent = Me
        fenetre3.Show()
```



Pour obtenir des fenêtres filles bien rangées dans leur fenêtre mère, vous pouvez faire appel à la méthode `LayoutMdi` en lui passant comme paramètre l'une des constantes prédéfinies de l'énumération `MdiLayout` :

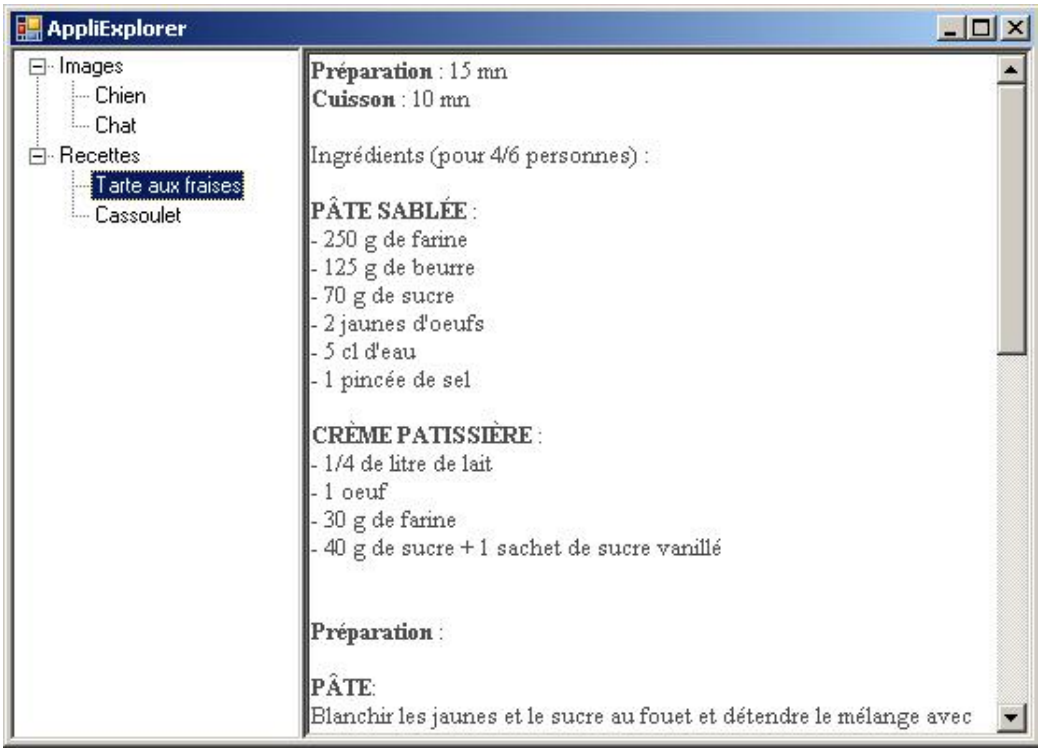
```
Me.LayoutMdi(MdiLayout.TileHorizontal)
```


```
Me.LayoutMdi(MdiLayout.TileVertical)
```


```
Me.LayoutMdi(MdiLayout.Cascade)
```

Ces différentes méthodes sont, en général, appelées par un menu de l'application qui fournit la liste des fenêtres ouvertes dans l'application. Nous verrons comment mettre cela en œuvre dans la section consacrée aux menus.

Pour illustrer une autre manipulation possible des fenêtres MDI, nous allons les utiliser pour réaliser une application de style Explorateur. Voici ci-dessous l'aspect général de l'application.



Dans la partie gauche, toujours visible, nous avons sous forme d'arborescence les documents disponibles dans l'application. En fonction de la sélection faite dans cette arborescence, la zone de droite s'adapte pour afficher soit l'image soit le texte d'une recette. Nous avons donc besoin de trois fenêtres différentes :

- la fenêtre principale qui va contenir le contrôle TreeView, et par la suite les fenêtres chargées de visualiser les documents,
- une fenêtre pour l'affichage des images,
- une fenêtre pour l'affichage du texte.

Préparons la fenêtre principale :

- Modifiez la propriété `IsMdiContainer` sur `True` pour activer la fonctionnalité de fenêtre mère MDI.
- Ajoutez un contrôle TreeView.
- Modifiez la propriété `Dock` du contrôle TreeView sur `Left` pour qu'il soit associé à la bordure gauche de la fenêtre.
- Ajoutez les éléments dans le contrôle TreeView en vous aidant de l'éditeur de nœuds.

- Pour notre application, la propriété `Name` des nœuds racines est utilisée pour déterminer le type de document (tx pour fichier texte, gr pour fichier graphique). Pour les autres nœuds de l'arborescence, elle mémorise le nom

du fichier concerné.

Les fenêtres enfants sont toutes aussi simples à configurer.

Pour la fenêtre graphique :

- Modifiez la propriété `BorderStyle` sur none.
- Ajoutez un contrôle PictureBox.
- Modifiez la propriété `dock` de ce contrôle sur Fill pour qu'il occupe toute la surface disponible de la fenêtre.
- Modifiez la propriété `SizeMode` de ce contrôle sur StretchImage pour que l'image s'adapte à la taille du contrôle (donc de la fenêtre).

Pour la fenêtre texte :

- Modifiez la propriété `BorderStyle` sur none.
- Ajoutez un contrôle RichTextBox.
- Modifiez la propriété `dock` de ce contrôle sur Fill pour qu'il occupe toute la surface disponible de la fenêtre.

Il ne nous reste plus maintenant qu'à écrire quelques lignes de code pour afficher la bonne fenêtre lors d'une sélection dans le contrôle TreeView. Voici ci-dessous ces quelques lignes de code.

```
Private Sub TreeView1_AfterSelect(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.TreeViewEventArgs) Handles TreeView1.AfterSelect
If Not IsNothing(e.Node.Parent) Then
     Dim f As Form
     For Each f In Me.MdiChildren
       f.Close()
     Next
  Select case e.Node.Parent.Name
  case "gr"
    Dim fGr As Graphique
    fGr = New Graphique
    fGr.MdiParent = Me
    fGr.Show()
    fGr.Dock = DockStyle.Fill
    fGr.PictureBox1.Image = Image.FromFile("../../" & e.Node.Name)
  case "tx"
    Dim fTx As Texte
    fTx = New Texte
    fTx.MdiParent = Me
    fTx.Show()
    fTx.Dock = DockStyle.Fill
    fTx.RichTextBox1.LoadFile("../../" & e.Node.Name)
End Select
End If
End Sub
```

La totalité du code se trouve dans la procédure événementielle `TreeView1_AfterSelect` appelée automatiquement après la sélection par l'utilisateur d'un élément dans le contrôle TreeView. Notre premier travail consiste à tester si c'est un nœud enfant qui vient d'être sélectionné. Si c'est le cas, nous fermons toutes les fenêtres enfants déjà présentes (toutes est un bien grand mot puisque avec ce mécanisme il ne pourra jamais y en avoir plus d'une d'affichée à la fois). Ensuite, nous testons le type de document demandé en vérifiant la propriété `Name` du nœud parent de l'élément sélectionné (gr ou tx). En fonction du résultat, nous créons une instance de la fenêtre adaptée à la situation. Nous établissons son lien de parenté avec la fenêtre principale (propriété `MdiParent`). La fenêtre est ensuite affichée en occupant toute la surface libre de la fenêtre mère Mdi (propriété `Dock`=DockStyle.Fill). Ultime étape, nous affichons le document dans le contrôle adapté, RichTextBox ou PictureBox.

Les événements clavier et souris

La gestion du clavier et de la souris s'effectue exclusivement en utilisant les différents événements que ces deux périphériques sont capables de déclencher. La plupart des contrôles sont capables de gérer ces événements, il nous faut simplement connaître à quel moment ils sont déclenchés et quelles informations ils fournissent.

1. Les événements clavier

Les possibilités d'action de l'utilisateur sur son clavier sont limitées : il peut simplement frapper une touche du clavier.

Dans Visual Basic.NET, cette action est décomposée en trois événements distincts :

`KeyDown`

Cet événement se produit lors de l'enfoncement de la touche.

`KeyUp`

Cet événement se produit lors du relâchement de la touche.

`KeyPress`

Cet événement se produit lors de l'enfoncement, uniquement si la touche correspond à un caractère ASCII.

Dans les événements `KeyDown` et `KeyUp`, un argument de type `KeyEventArgs` donne des informations complémentaires sur l'événement.

Les propriétés suivantes sont notamment disponibles :

- [Alt] indique l'état de la touche [Alt] du clavier au moment où l'événement se produit.
- [Ctrl] et [Shift] fournissent des informations identiques pour les touches [Ctrl] et [Shift].
- `KeyCode` indique le numéro de la touche sur le clavier.

L'événement `KeyDown` est utilisé principalement pour travailler avec les touches de fonction. Nous pouvons, par exemple, convertir en majuscules le texte d'un contrôle `TextBox` si l'utilisateur frappe la combinaison de touches [Shift] [Ctrl] [Alt] [F8].

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyDown
        If e.Alt And e.Control And e.Shift And (e.KeyCode = Keys.F8) Then
            TextBox1.Text = UCase(TextBox1.Text)
        End If
    End Sub
```

Une autre solution est possible en utilisant la propriété `Modifiers` qui regroupe l'état des touches [Shift], [Ctrl] et [Alt].

```
Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows. Forms.KeyEventArgs) Handles TextBox1.KeyDown
        If e.Modifiers = Keys.Shift + Keys.Control + Keys.Alt
And e.KeyCode = Keys.F8 Then
            TextBox1.Text = UCase(TextBox1.Text)
        End If
    End Sub
```

Enfin, la dernière possibilité consiste à utiliser la propriété `KeyData` qui regroupe l'état des touches [Shift], [Ctrl], [Alt] et une touche de fonction.

```
  Private Sub TextBox1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles TextBox1.KeyDown
If e.KeyData = Keys.Shift + Keys.Control + Keys.Alt + Keys.F8
Then
```

```
            TextBox1.Text = UCase(TextBox1.Text)
        End If
    End Sub
```

L'événement KeyPress nous informe simplement, par l'intermédiaire du paramètre e de typeKeyPressEventArgs, sur le caractère qui vient d'être saisi. Nous pouvons par exemple utiliser cet événement pour vérifier que tous les caractères saisis sont numériques, si ce n'est pas le cas, nous émettons un beep.

```
  Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
        If Not IsNumeric(e.KeyChar) Then
            Beep()
        End If
    End Sub
```

Cette procédure informe simplement l'utilisateur qu'il a saisi un caractère invalide, mais le caractère apparaît tout de même dans la zone de texte. Il serait plus judicieux d'éviter que le caractère atteigne la zone de texte. La propriété Handled du paramètre permet, en lui affectant la valeur **true**, d'indiquer que l'événement clavier vient d'être géré et qu'il ne doit pas être pris en compte par le contrôle.

```
  Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
        If Not IsNumeric(e.KeyChar) Then
            Beep()
            e.Handled = True
        End If
    End Sub
```

Si plusieurs contrôles TextBox doivent avoir le même fonctionnement, vous pouvez recopier ce code dans les événements KeyPress de chacun des contrôles. Une solution plus élégante nous permet de centraliser le traitement dans l'événement KeyPress de la fenêtre. Pour cela, il convient d'indiquer à la propriétéKeyPreview de la fenêtre qu'elle recevra les événements clavier avant le contrôle de la fenêtre qui a le focus. Il faut simplement tester le contrôle actif de la feuille et vérifier s'il doit "subir" le traitement.

```
  Private Sub TextBox1_KeyPress(ByVal sender As Object, ByVal e As System.
Windows.Forms.KeyPressEventArgs) Handles TextBox1.KeyPress
        If Me.ActiveControl Is TextBox1 Then
            If Not IsNumeric(e.KeyChar) Then
                Beep()
                e.Handled = True
            End If
        End If
    End Sub
```

2. Les événements souris

Ces événements sont principalement liés à l'utilisation des boutons ou de la molette de la souris. L'événementclick est le plus utilisé des événements souris. Il correspond à un appui suivi d'un relâchement du bouton principal de la souris. Il faut bien parler de bouton principal de la souris car en fonction de la configuration du poste de travail, ce bouton principal peut être aussi le bouton droit (configuration pour gaucher). L'événementDoubleClick n'est pas pris en charge par tous les contrôles, ainsi, par exemple, les contrôles Button ne sont pas capables de gérer le double clic, ils génèrent en fait deux événements click à la suite.

Des événements souris plus élémentaires sont aussi disponibles :

- MouseDownlorsqu'un bouton de la souris est enfoncé ;
- MouseUplorsqu'un bouton de la souris est relâché ;
- MouseWheellorsque la molette est actionnée.

Pour ces trois événements, un paramètre de type MouseEventArgs est fourni. Par l'intermédiaire des propriétés disponibles dans cette classe, on obtient les informations suivantes :

- le bouton à l'origine de l'événement avec la propriété Button ;

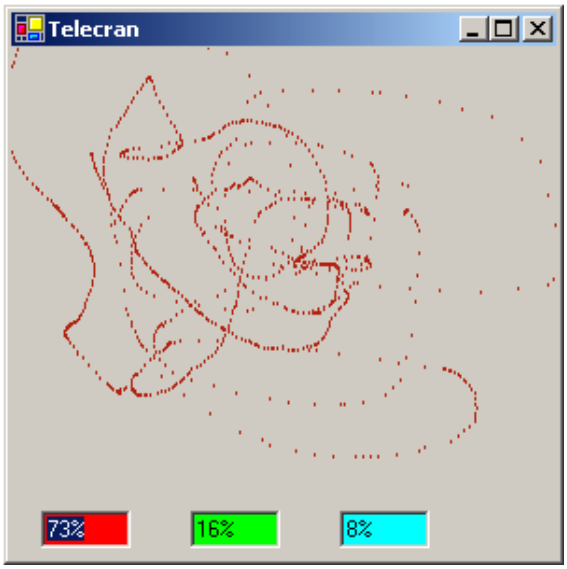
- le nombre de fois où le bouton est enfoncé ou relâché avec la propriété `Clicks` ;
- le nombre de crans de déplacement de la roulette avec la propriété `Delta`. Cette propriété est un entier positif ou négatif, suivant le sens de rotation de la molette. Chaque déplacement de la molette d'un cran incrémente ou décrémente cette propriété d'une valeur de 120 ;
- les propriétés x et y indiquent l'emplacement sur le contrôle où l'événement souris vient de se produire.

Les déplacements de la souris génèrent quatre événements :

- `MouseEnter`lorsque la souris entre au-dessus d'un contrôle ;
- `MouseMove`lorsque la souris se déplace sur le contrôle ;
- `MouseLeave`lorsque la souris quitte la surface du contrôle ;
- `MouseHover`lorsque la souris reste au-dessus d'un contrôle pendant une seconde.

Comme pour les événements liés aux boutons de la souris, un paramètre de type `Mouse EventArgs`nous est fourni dans ces événements.

Un petit exemple pour tester cela avec une application de dessin très rudimentaire.



À chaque déplacement de la souris, un point est tracé sur la feuille avec la couleur indiquée par les trois zones de texte (rouge, vert, bleu). Pour modifier la couleur, il suffit de déplacer la souris au-dessus de la zone de texte correspondante (le focus se place automatiquement grace à l'événement`MouseHover`) et d'actionner la molette de la souris pour incrémenter ou décrémenter le pourcentage.

```
Public Class telecran
     Dim rouge, vert, bleu As Integer
     Private Sub telecran_MouseMove(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles Me.MouseMove
        Dim g As Graphics
        Dim crayon As Pen
        If e.Button = MouseButtons.Left Then
           g = Graphics.FromHwnd(Me.Handle)
           crayon = New Pen(Color.FromArgb(rouge * 2.55, vert * 2.55, bleu * 2.55))
           g.DrawEllipse(crayon, e.X, e.Y, 1, 1)
        End If
    End Sub
    Private Sub txtRouge_MouseWheel(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles txtRouge.MouseWheel
        rouge += e.Delta / 120
        If rouge > 100 Then
```

```
            rouge = 100
            txtRouge.Text = "100%"
        ElseIf rouge < 0 Then
            rouge = 0
            txtRouge.Text = "0%"
        Else
            txtRouge.Text = rouge & "%"
        End If
    End Sub
    Private Sub txtVert_MouseWheel(ByVal sender As Object, ByVal e As
System.Windows.Forms.MouseEventArgs) Handles txtVert.MouseWheel
        vert += e.Delta / 120
        If vert > 100 Then
            vert = 100
            txtVert.Text = "100%"
        ElseIf rouge < 0 Then
            vert = 0
            txtVert.Text = "0%"
        Else
            txtVert.Text = vert & "%"
        End If
    End Sub

    Private Sub txtBleu_MouseWheel(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.MouseEventArgs) Handles txtBleu.MouseWheel
        bleu += e.Delta / 120
        If bleu > 100 Then
            bleu = 100
            TxtBleu.Text = "100%"
        ElseIf bleu < 0 Then
            bleu = 0
            TxtBleu.Text = "0%"
        Else
            TxtBleu.Text = bleu & "%"
        End If
    End Sub
    Private Sub txtRouge_MouseHover(ByVal sender As Object, ByVal e As
System.EventArgs) Handles txtRouge.MouseHover
        txtRouge.Focus()
    End Sub

    Private Sub txtVert_MouseHover(ByVal sender As Object, ByVal e As
System.EventArgs) Handles txtVert.MouseHover
        txtVert.Focus()
    End Sub

    Private Sub txtBleu_MouseHover(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles txtBleu.MouseHover
        TxtBleu.Focus()
    End Sub
End Class
```

3. Le Drag and Drop

Le Drag and Drop est une fonctionnalité très utilisée dans les applications Windows. Elle permet le déplacement d'informations dans une application ou entre applications, en "accrochant" un élément au curseur de la souris et en le lâchant ensuite au-dessus de sa destination.

On peut pratiquement accrocher n'importe quel élément au curseur de la souris pour ensuite le déplacer ou le copier (texte, image, fichier...). Pour guider l'utilisateur sur ces possibilités, le curseur de la souris est modifié en fonction des contrôles qui sont survolés par la souris.

Une opération de Drag and Drop se déroule en trois étapes :

- accrochage d'un élément à la souris ;
- déplacement de la souris vers la destination ;

- largage au-dessus de la destination.

Lors de l'opération de largage, nous avons deux possibilités :

- l'élément accroché à la souris est déplacé. Dans ce cas, il disparaît du contrôle de départ au moment du largage au-dessus du contrôle de destination.

- L'élément accroché est copié. Dans ce cas, c'est une copie qui est larguée au-dessus du contrôle de destination.

Regardons en détail le code nécessaire pour mener à bien une opération de Drag and Drop. Nous prendrons un exemple simple réalisant une copie entre deux zones de texte (`TxtSource`, `TxtDestination`).

a. Démarrage du Drag and Drop

En général, une opération de Drag and Drop est démarrée lorsque la souris se déplace au- dessus d'un contrôle et le sélectionne sans relâchement du bouton de la souris. On considère, dans ce cas, que l'utilisateur vient d'accrocher le contrôle à sa souris.

Dans le code, cela ce traduit simplement par l'appel de la méthode `DoDragDrop` du contrôle de départ. L'appel de cette méthode nécessite deux paramètres :

- L'élément que nous accrochons au curseur de la souris (une chaîne de caractères, une image bitmap ou une image métafile),

- Les opérations autorisées pour l'objet accroché (copie, déplacement...).

```
Private Sub txtsource_MouseMove(ByVal sender As Object, ByVal e As System.
Windows.Forms.MouseEventArgs) Handles txtsource.MouseMove
   If e.Button = MouseButtons.Left Then
            txtsource.DoDragDrop(txtsource.Text, DragDropEffects.Move +
DragDropEffects.Copy)
   End If
End Sub
```

b. Configuration des contrôles pour la réception

Il convient avant tout de configurer les contrôles de destination pour qu'ils acceptent la réception d'un élément en modifiant la propriété`AllowDrop` sur **true**.

Ensuite, nous devons gérer l'événement`DragEnter` qui se produit lorsque la souris entre sur la surface du contrôle avec un élément "accroché". Dans la gestion de cet événement, il faut déterminer ce qui est accroché au curseur de la souris pour savoir si le contrôle peut l'accepter. Enfin, il faut déterminer si l'utilisateur souhaite faire un déplacement ou une copie de l'élément accroché au curseur de la souris.

Pour effectuer cela dans le code, dans l'événement `DragEnter`, nous avons à notre disposition le paramètre e qui est une instance de la classe `DragEventArg`. Cette classe nous fournit de nombreuses propriétés pour nous aider dans nos décisions :

`AllowedEffect`

Permet de savoir quelles sont les opérations autorisées par le contrôle à l'origine du Drag and Drop.

`Data`

Contient les informations accrochées au curseur de la souris. Par l'intermédiaire de cette propriété, on peut obtenir le type d'information en appelant la méthode `GetDataPresent` ou obtenir les données en appelant la méthode `GetData`.

`Effect`

Indique l'action autorisée par le contrôle de destination. Cette propriété est utilisée pour contrôler l'apparence du curseur de la souris.

`KeyState`

Indique l'état des touches [Shift], [Ctrl], [Alt] nous permettant de savoir si l'utilisateur souhaite effectuer un déplacement ou une copie.

`X, Y`

Indique les coordonnées de la souris sur le contrôle.

```
Private Sub txtdestination_DragEnter(ByVal sender As Object, ByVal e As System.
Windows.Forms.DragEventArgs) Handles txtdestination.DragEnter
        ‘qu’est ce qu’il y a d’accroché ?
        If e.Data.GetDataPresent(DataFormats.Text) Then
            ‘ est ce que la touche Ctrl est enfoncée
            If (e.KeyState And 8) = 8 Then
                ‘ curseur copie
                e.Effect = DragDropEffects.Copy
            Else
                ‘ curseur souris déplacement
                e.Effect = DragDropEffects.Move
            End If
        End If
End Sub
```

c. Récupération de l'élément accroché

Lorsque l'utilisateur relâche le bouton de la souris, l'événement `DragDrop` se produit sur le contrôle de destination. Dans cet événement, il faut récupérer l'élément accroché et le placer dans le contrôle de destination. S'il s'agit d'un déplacement, il faut aussi supprimer l'information du contrôle source. Problème ! qui est le contrôle source ? Nous n'avons aucune information sur son identité. La solution consiste à stocker, dans une variable une référence vers le contrôle à l'origine de l'opération de Drag and Drop au début de l'opération.

```
Dim ctrlSource As TextBox

Private Sub txtsource_MouseMove(ByVal sender As Object, ByVal e As System.
Windows.Forms.MouseEventArgs) Handles txtsource.MouseMove
        If e.Button = MouseButtons.Left Then
            ctrlSource = txtsource
            txtsource.DoDragDrop(txtsource.Text, DragDropEffects.Move + Drag
DropEffects.Copy)
        End If
End Sub
Private Sub txtdestination_DragDrop(ByVal sender As Object, ByVal e As System.
Windows.Forms.DragEventArgs) Handles txtdestination.DragDrop
      ‘ récupération de l’info est stockage dans le controle de destination
        txtdestination.Text = e.Data.GetData(DataFormats.Text)
      ‘ vérification si c’est une copie ou un déplacement
        If (e.KeyState And 8)  8 Then
            ‘ effacement du controle source
            ctrlSource.Clear()
        End If
End Sub
```

Les boîtes de dialogue

Les boîtes de dialogue sont des fenêtres qui ont une fonction spéciale dans une application. Elles sont, en général, utilisées pour demander la saisie d'informations à l'utilisateur. Pour s'assurer que ces informations sont bien saisies avant de continuer l'exécution de l'application, les boîtes de dialogue sont souvent affichées en mode modal, c'est-à-dire que le reste de l'application est bloqué tant que la boîte de dialogue est affichée. Il arrive fréquemment que dans une application, on ait besoin des mêmes informations : un nom de fichier à ouvrir, une police de caractère à choisir, etc. Pour nous éviter d'avoir à recréer, à chaque fois, une nouvelle boîte de dialogue, nous avons à notre disposition une série de boîtes de dialogue prédéfinies.

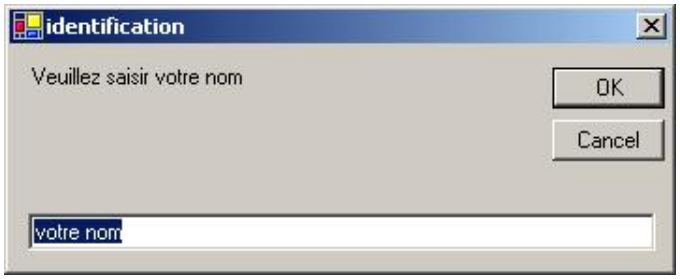
1. La boîte de saisie

La boîte de saisie permet de demander à l'utilisateur la saisie d'une chaîne de caractères. Cette possibilité est accessible par l'intermédiaire de la fonction`InputBox`. Cette fonction attend trois paramètres :

- le texte affiché dans la boîte (en général la question posée à l'utilisateur) ;
- le titre de la boîte de dialogue ;
- une valeur par défaut affichée dans la zone de saisie.

La fonction renvoie en retour une chaîne de caractères correspondant au texte saisi par l'utilisateur, si celui-ci valide sa saisie par le bouton **OK** sinon elle renvoie une chaîne de caractères vide.

```
Dim resultat As String
resultat = InputBox("Veuillez saisir votre nom", "identification", "votre nom")
```



2. La boîte de message

Les boîtes de message permettent de passer une information à l'utilisateur et lui donnent la possibilité de répondre par l'intermédiaire des boutons de commande de la boîte de message.

La boîte de message est disponible par l'intermédiaire de la méthode `show` disponible dans la classe `MessageBox`. Cette méthode prend de nombreux paramètres pour configurer la boîte de dialogue. Le premier paramètre correspond au message affiché. Le paramètre suivant spécifie le titre de la boîte de message. Les paramètres suivants doivent être choisis parmi des constantes prédéfinies pour indiquer respectivement :

- les boutons disponibles sur la boîte de message ;
- l'icône affichée sur la boîte de message ;
- le bouton sélectionné par défaut à l'affichage de la boîte de message.

Les constantes disponibles sont :

- pour les boutons :

Constante	Signification

OK	Bouton OK seul
OKCancel	Boutons OK et Annuler
AbortRetryIgnore	Boutons Abandonner, Réessayer et Ignorer
YesNoCancel	Boutons Oui, Non et Annuler
YesNo	Boutons Oui et Non
RetryCancel	Boutons Réessayer et Annuler

- pour les icônes :

Constante	Signification
IconInformation	
IconExclamation	
IconError	
IconQuestion	

- pour le bouton par défaut :

Constante	Signification
DefaultButton1	Premier bouton
DefaultButton2	Deuxième bouton
DefaultButton3	Troisième bouton

Pour obtenir la boîte de message suivante,

nous utiliserons le code suivant :

```
Dim reponse As Integer
MessageBox.Show("Voulez vous enregistrer en quittant l'application", _
                "Fin de programme", MessageBoxButtons.YesNoCancel, _
                MessageBoxIcon.Question,MessageBoxDefaultButton.Button1)
```

Comme nous posons une question à l'utilisateur, nous devons récupérer sa réponse pour décider de la conduite à tenir dans l'application. Pour cela, la méthode `Show` renvoie une valeur indiquant le bouton utilisé pour fermer la boîte de message. Ici encore, une série de constantes est définie pour identifier chaque cas possible.

Valeur renvoyée	Bouton utilisé
Ok	Bouton Ok
Cancel	Bouton Annuler
Abort	Bouton Abandonner
Retry	Bouton Réessayer
Ignore	Bouton Ignorer
Yes	Bouton Oui
No	Bouton Non

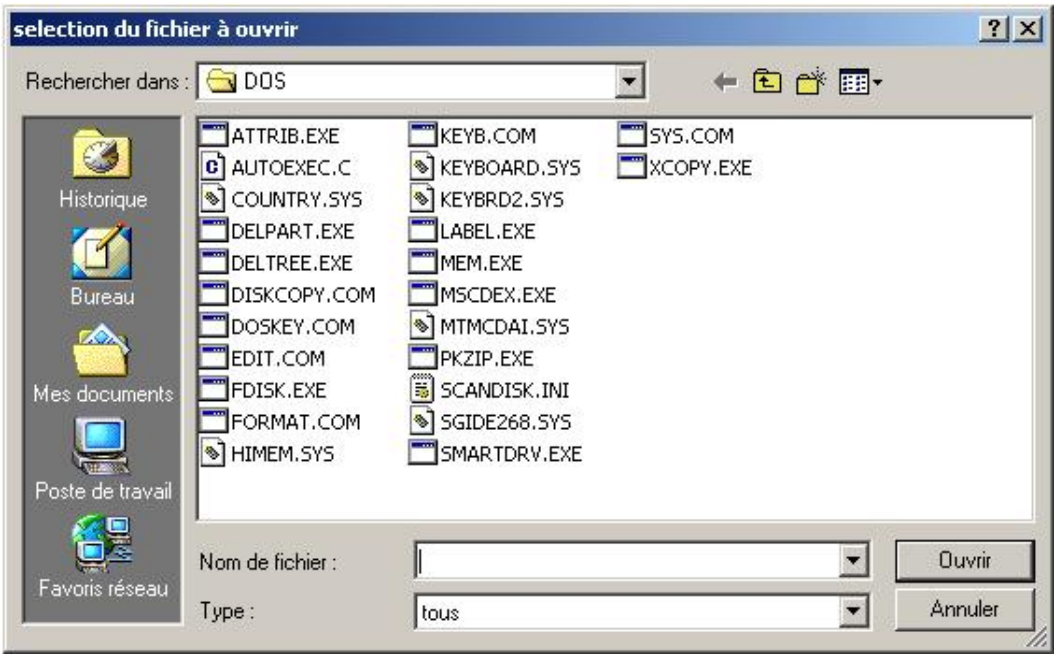
Nous pouvons ensuite tester la réponse :

```
Select Case reponse
    Case DialogResult.Yes
        ...
    Case DialogResult.No
        ...
    Case DialogResult.Cancel
        ...
End Select
```

3. Les boîtes de dialogue de Windows

De nombreuses boîtes de dialogue sont déjà définies au niveau du système Windows lui-même. Pour pouvoir les utiliser dans nos applications, nous avons à notre disposition une série de classes. Regardons comment les configurer et les utiliser dans une application.

a. Dialogue d'ouverture de fichier



Cette boîte de dialogue nous permet la sélection d'un ou de plusieurs noms de fichier avec la possibilité de se déplacer dans l'arborescence de la machine. La classe utilisée est la classe`OpenFileDialog`. Nous devons donc créer une instance dans notre application.

```
Dim dlgOuvrir As OpenFileDialog
dlgOuvrir = New OpenFileDialog()
```

Il convient également de configurer notre boîte de dialogue. La propriété`InitialDirectory` indique le répertoire sur lequel se trouve la boîte de dialogue à son ouverture. Il est possible de n'afficher que certains fichiers dans les répertoires qui seront parcourus, il faut donc configurer par l'intermédiaire de la propriété `Filter` les correspondances entre la description du contenu et l'extension associée. La propriété `Filter` stocke l'information sous forme d'une chaîne de caractères. La description et l'extension sont séparées dans la chaîne par le caractère | ([AltGr] 6). Si plusieurs extensions sont disponibles pour une même description, elles doivent être séparées par un point-virgule dans la chaîne. Vous pouvez également indiquer si une extension doit être ajoutée aux noms de fichiers saisis manuellement, si ceux-ci n'en comportent pas.

La propriété`DefaultExt` contient l'extension à ajouter et`AddExtension` indique si cette extension est ajoutée automatiquement. Dans la mesure où l'on permet à l'utilisateur de saisir manuellement le chemin et le nom du fichier à ouvrir, vous pouvez confier à la boîte de dialogue le soin de vérifier que le nom et le chemin d'accès sont corrects. Les propriétés `CheckFileExist`et`CheckPathExist` gèrent ces vérifications. Vous pouvez également autoriser les sélections multiples par l'intermédiaire de la propriété`Multiselect`.

Enfin, pour afficher la boîte de dialogue, on utilise la méthode `ShowDialog` :

```
      dlgOuvrir.InitialDirectory = "c:\dos"
      dlgOuvrir.Title = "selection du fichier à ouvrir"
      dlgOuvrir.Filter ="tous|*.*|Images|*.bmp;*.gif;*.jpg|texte|*.txt"
      dlgOuvrir.DefaultExt = "toto"
      dlgOuvrir.AddExtension = True
      dlgOuvrir.CheckFileExists = False
      dlgOuvrir.Multiselect = True
 dlgOuvrir.ShowDialog()
```

Les noms du fichier ou des fichiers sélectionnés sont disponibles dans la propriété`FileNames` pour une sélection unique ou dans la propriété `FileNames` pour les sélections multiples. Cette propriété `FileNames` est un tableau de chaînes de caractères avec, dans chaque case du tableau, le nom complet d'un des fichiers sélectionnés.

```
       Dim nomFichier As String
       For Each nomFichier In dlgOuvrir.FileNames
           System.Console.WriteLine(nomFichier)
  Next
```

b. Dialogue d'enregistrement de fichier

La boîte de dialogue d'enregistrement de fichier est similaire à la précédente, mis à part la propriété `Multiselect` qui disparaît et les propriétés`CreatePrompt` et`OverwritePrompt` permettant d'afficher un message d'avertissement, si le nom du fichier saisi n'existe pas ou au contraire s'il existe déjà.

c. Dialogue de choix de répertoire

Cette boîte de dialogue est utilisée pour la sélection ou la création d'un répertoire. Elle est créée à partir de la classe

`FolderBrowserDialog`. Cette dernière comporte très peu de propriétés. La plus utilisée est certainement la propriété `SelectedPath` permettant la récupération du chemin d'accès au répertoire sélectionné. Le répertoire racine de la boîte de dialogue est indiqué par la propriété `RootFolder`. Cette propriété reçoit une des valeurs de l'énumération `Environment.SpecialFolder` représentant les principaux répertoires caractéristiques du système comme, par exemple, le répertoire **Mes documents**. Si cette propriété est utilisée la sélection ne pourra se faire que dans un sous répertoire du répertoire racine. L'ajout d'un bouton permettant la création d'un nouveau répertoire peut être autorisé en modifiant la propriété `ShowNewFolderButton`. L'affichage de la boîte de dialogue se fait de manière classique par la méthode `ShowDialog` :

```
dlgChoixRep.RootFolder = Environment.SpecialFolder.MyDocuments
dlgChoixRep.ShowDialog()
MsgBox(dlgChoixRep.SelectedPath, , "Répertoire sélectionné")
```

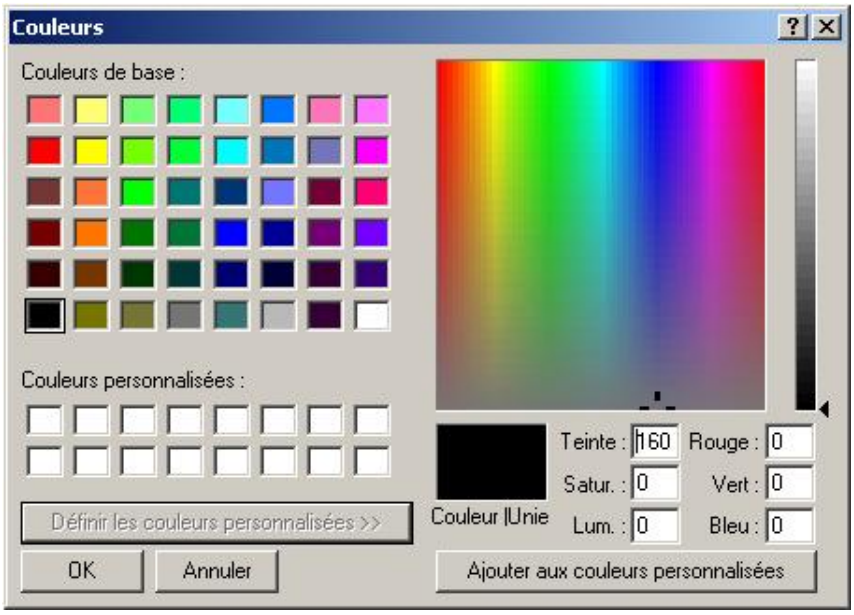
Il faut également noter que le chemin d'accès retourné par cette boîte de dialogue est un chemin absolu comme dans l'exemple ci-dessous :

d. Dialogue de choix d'une couleur

La boîte de dialogue de choix de couleur créée à partir de la classe `ColorDialog` peut avoir deux configurations différentes.

Une version "simple" où seules les couleurs de base sont disponibles.

Une version complète dans laquelle l'utilisateur pourra créer des couleurs personnalisées.



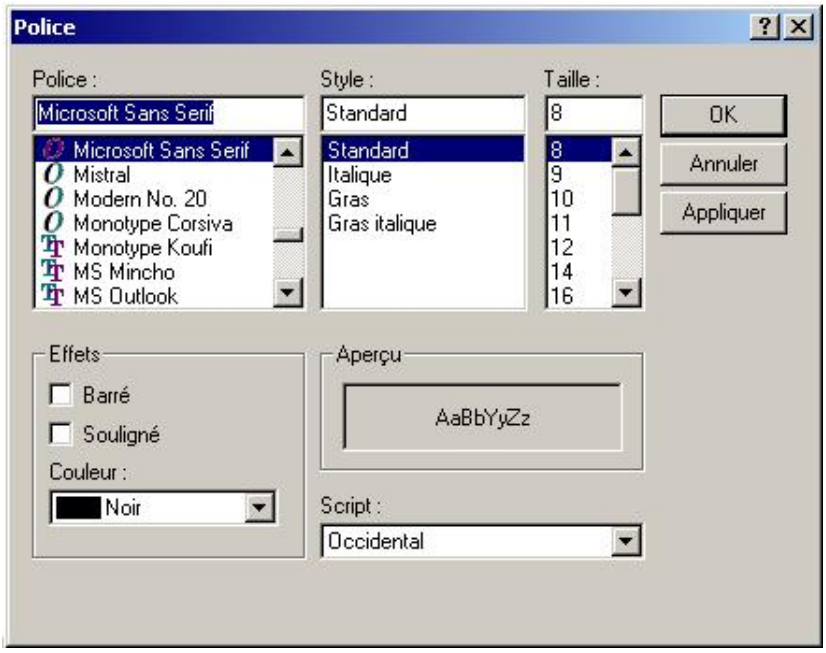
La propriété `Color` permet d'initialiser la boîte de dialogue avant son affichage et ensuite de récupérer la couleur choisie par l'utilisateur. Vous pouvez interdire l'utilisation des couleurs personnalisées ou au contraire afficher la boîte de dialogue complète dès son ouverture. Pour interdire l'affichage des couleurs personnalisées, on modifie la propriété `AllowFullOpen`. Pour forcer l'affichage complet, on utilise la propriété `FullOpen`.

L'affichage de la boîte de dialogue s'effectue toujours par la méthode `ShowDialog`. Pour conserver une qualité d'affichage correcte, vous pouvez également n'autoriser que l'utilisation de couleurs pures (les couleurs obtenues par juxtaposition de différents pixels seront éliminées des choix possibles). Cette option est à utiliser si vous avez une carte graphique configurée en 256 couleurs.

Cet exemple modifie la couleur de fond de notre feuille.

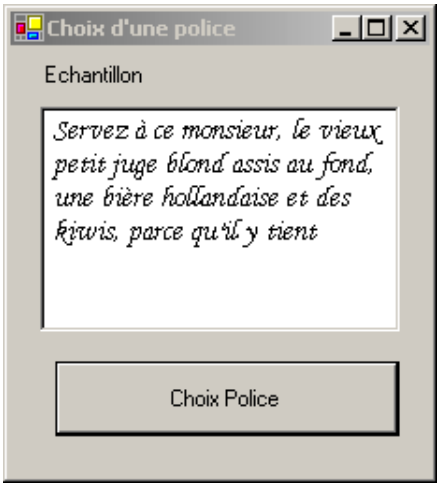
```
Dim dlgColor As ColorDialog
dlgColor = New ColorDialog()
dlgColor.FullOpen = True
dlgColor.SolidColorOnly = True
dlgColor.Color = Me.BackColor
dlgColor.ShowDialog()
Me.BackColor = dlgColor.Color
```

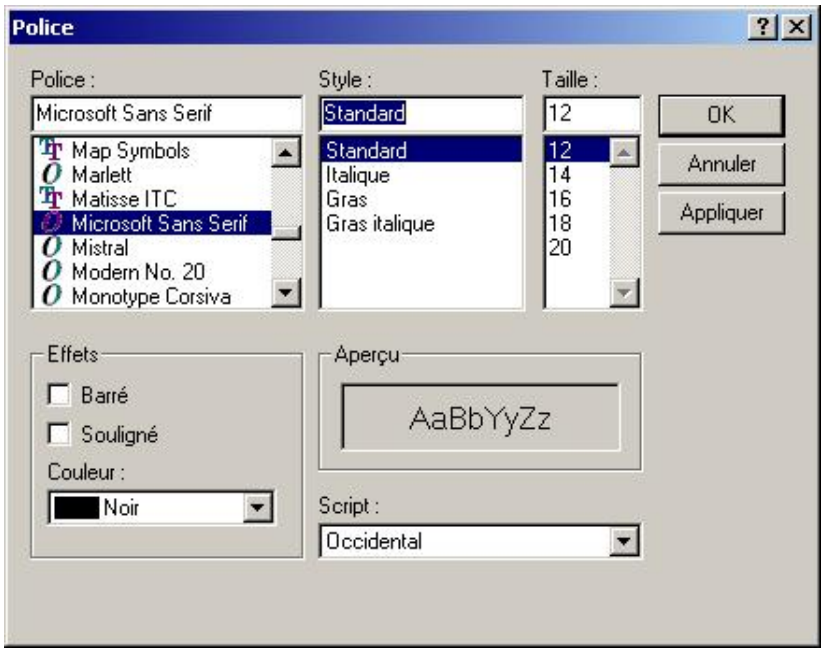
e. Dialogue de choix d'une police



La classe de base utilisée pour la sélection d'une police est la classe `FontDialog`. La propriété `Font` permet de définir la police de caractères utilisée pour initialiser la boîte de dialogue ou, après fermeture, de récupérer la police sélectionnée. Vous pouvez également afficher une boîte de dialogue simplifiée sans le choix de couleur ou des effets. Pour cela, les propriétés `ShowColor` et `ShowEffects` contrôlent l'affichage de ces paramètres dans la boîte de dialogue. Afin de garantir que les paramètres sélectionnés correspondent bien à une police existant sur la machine, vous pouvez utiliser la propriété `FontMustExist`. Cette propriété obligera la boîte de dialogue à vérifier l'existence d'une police correspondante sur le système avant de se fermer. Certaines polices proposent plusieurs jeux de caractères. Vous pouvez autoriser les utilisateurs à choisir l'un de ces jeux de caractères en modifiant la propriété `AllowScriptChange`. La taille de la police sélectionnée peut également être limitée par les propriétés MaxSize et`MinSize`.

Pour vous rendre compte de l'effet de la police sélectionnée, un aperçu sur quelques caractères est disponible. Si cet aperçu n'est pas suffisant, vous avez la possibilité d'afficher un bouton **Appliquer** sur votre boîte de dialogue par l'intermédiaire de la propriété `ShowApply`. Ce bouton déclenche un événement `Apply` sur la boîte de dialogue. Dans la gestion de cet événement, vous pouvez utiliser la propriété `Font` de la boîte de dialogue pour visualiser l'effet de la police actuellement sélectionnée sur votre texte. La variable faisant référence à la boîte de dialogue doit être déclarée avec le mot clé `WithEvents` donc en dehors d'une procédure. Un petit exemple pour visualiser l'utilisation de ces propriétés :



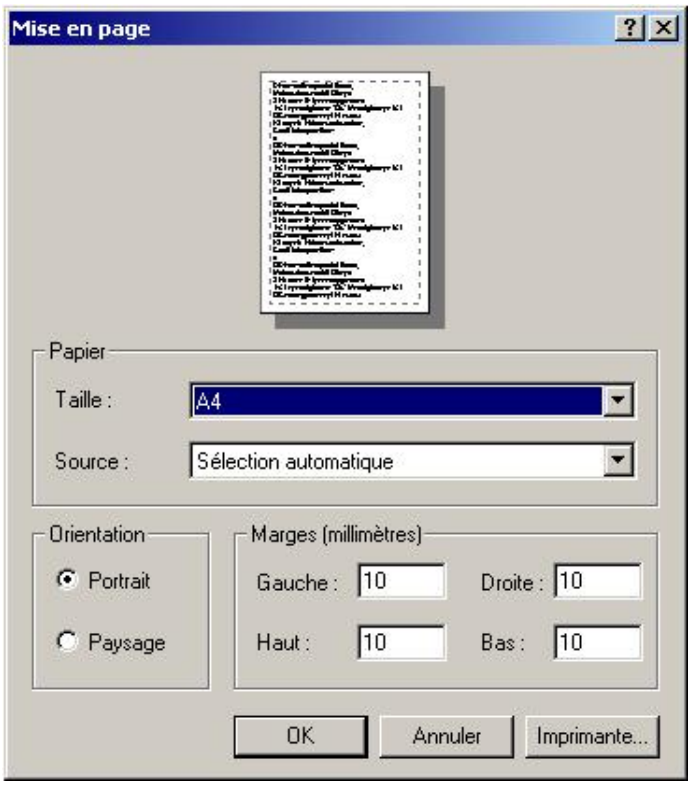



```
Public Class TestPolice
    Inherits System.Windows.Forms.Form
    Dim WithEvents dlgFont As FontDialog
Private Sub cmdPolice_click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs)Handles cmdPolice.Click
        dlgFont = New FontDialog()
        dlgFont.ShowApply = True
        dlgFont.ShowColor = True
```

```
        dlgFont.ShowEffects = True
        dlgFont.MaxSize = 20
        dlgFont.MinSize = 12
        dlgFont.FontMustExist = True
        dlgFont.AllowScriptChange = True
        dlgFont.ShowDialog()
        txtEchantillon.Font = dlgFont.Font
End Sub
Private Sub dlgFont_Apply(ByVal sender As Object, ByVal e As System.EventArgs)
     Handles dlgFont.Apply
        txtEchantillon.Font = dlgFont.Font
    End Sub
End Class
```

f. Dialogue de mise en page

Par l'intermédiaire de cette boîte de dialogue, vous allez pouvoir configurer les paramètres de mise en page de votre document (marges, orientation...).



Cette boîte de dialogue est créée à partir de la classe`PageSetupDialog`. Pour travailler, cette classe a besoin de deux classes auxiliaires : la classe`PageSettings` sert à stocker la configuration de la mise en page, la classe `PrinterSettings` stocke la configuration de l'imprimante sélectionnée. Il faut créer une instance de ces deux classes et les associer aux propriétés `PageSettings` et`PrinterSettings` de la boîte de dialogue. Vous serez obligé d'importer l'espace de noms `System.Drawing.Printing` pour pouvoir utiliser ces deux classes.

L'utilisation des différentes rubriques de la boîte de dialogue peut être interdite par la modification des propriétés suivantes :

- `AllowMargins` pour la modification des marges ;
- `AllowOrientation` pour la modification de l'orientation ;
- `AllowPaper` pour le choix du papier ;
- `AllowPrinter` pour le choix de l'imprimante.

Les choix de l'utilisateur seront ensuite récupérés à l'aide des propriétés `PageSettings` et `PrinterSettings` de la boîte

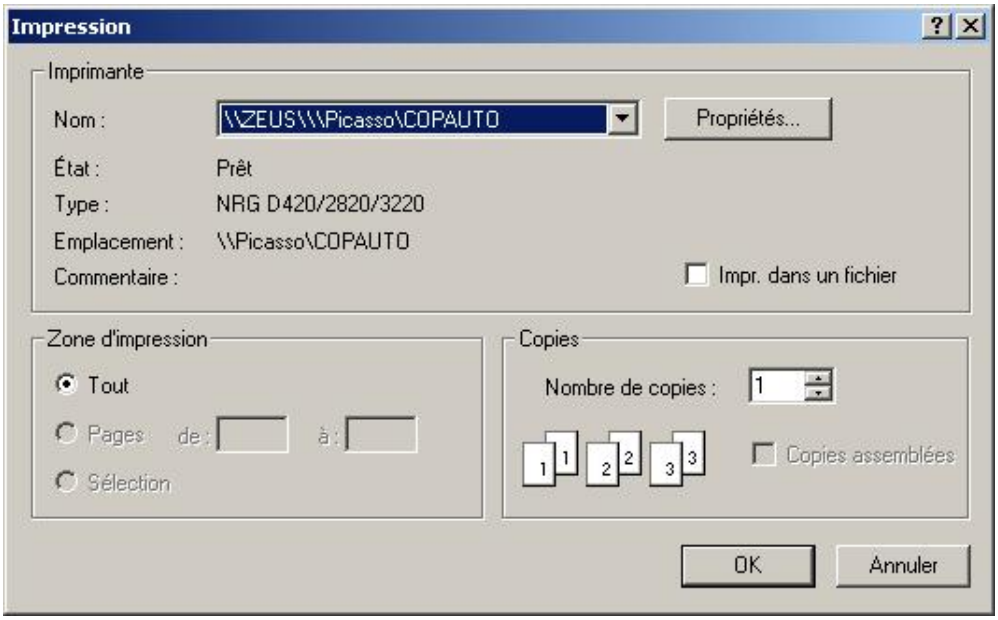
de dialogue.

Voici un exemple d'utilisation :

```
Imports System.Drawing.Printing
Public Class Form2
    Inherits System.Windows.Forms.Form

Private Sub TestPgSetup_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles testPgSetup.Click
        Dim configPg As PageSettings
        Dim configPrt As PrinterSettings
        configPg = New PageSettings()
        configPrt = New PrinterSettings()
        dlgPgSetup.PageSettings() = configPg
        dlgPgSetup.AllowPrinter = True
        dlgPgSetup.PrinterSettings = configPrt
        dlgPgSetup.ShowDialog()
        MessageBox.Show("vous avez choisi d'imprimer avec l'imprimante " _
                        & dlgPgSetup.PrinterSettings.PrinterName _
                        & " sur du papier " _
   & dlgPgSetup.PageSettings.PaperSize.PaperName _
                        & " en format " _
&(IIf(dlgPgSetup.PageSettings.Landscape, "paysage", "portrait")))
    End Sub
End Class
```

g. Dialogue de configuration d'impression

Avec cette boîte de dialogue, vous pouvez configurer les paramètres d'impression de votre document. Elle sera créée à partir de la classe`PrintDialog`.



Comme pour la boîte de dialogue de mise en page, la boîte de dialogue de configuration d'impression a besoin d'une instance de la classe `PrinterSettings` pour stocker les informations de configuration de l'imprimante.

Les différentes rubriques peuvent être interdites d'utilisation, par la modification des propriétés suivantes :

- `AllowSelection` autorise l'utilisation du bouton **Sélection**. En général ce bouton est accessible uniquement s'il y a quelque chose de sélectionné dans le document que vous voulez imprimer.
- `AllowSomePages` autorise la sélection d'une page de début et d'une page de fin pour l'impression du document. Ce bouton doit être disponible si le document contient plusieurs pages.
- `AllowPrintToFile` indique si la case à cocher **Impression dans un fichier** est disponible. Cette fonctionnalité

permet par exemple la récupération d'un fichier au format PostScript pour l'importer dans une autre application.

Le résultat des différentes options sélectionnées est disponible après fermeture de la boîte de dialogue, par l'intermédiaire de la propriété `PrinterSettings`.

Voici un nouvel exemple pour cette boîte de dialogue.

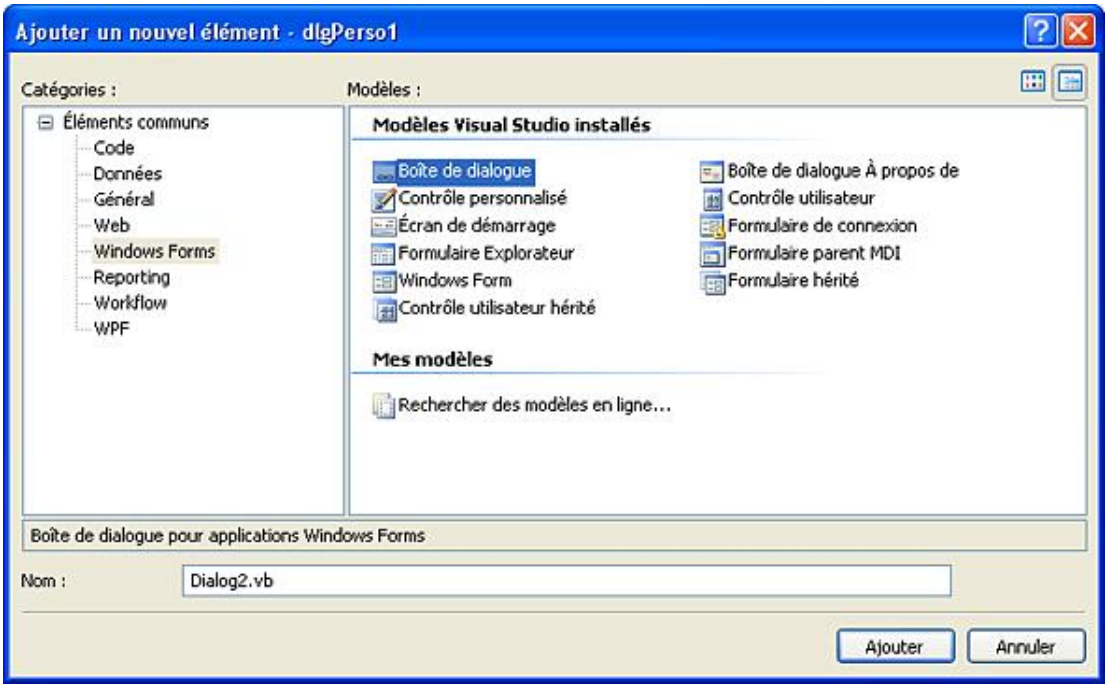
```
Dim configPrt As PrinterSettings
Dim dlgprinter As PrintDialog
configPrt = New PrinterSettings()
dlgprinter = New PrintDialog()
dlgprinter.PrinterSettings = configPrt
dlgprinter.AllowSomePages = True
dlgprinter.AllowSelection = True
dlgprinter.ShowDialog()
Select Case dlgprinter.PrinterSettings.PrintRange
  Case PrintRange.AllPages
MessageBox.Show("vous avez demandé l'impression de tout le document")
       Case PrintRange.SomePages
     MessageBox.Show("vous avez demandé l impression de la page " _
                & dlgprinter.PrinterSettings.FromPage & " à la page " _
                & dlgprinter.PrinterSettings.ToPage)
       Case PrintRange.Selection
     MessageBox.Show("vous avez demandé l'impression de la sélection")
End Select
```

4. Boîte de dialogue personnalisée

Après ce bref aperçu des boîtes de dialogue prédéfinies, nous allons voir comment créer nos propres boîtes de dialogue. La base de création d'une boîte de dialogue est une fenêtre classique pour laquelle on modifie les propriétés suivantes :

- Le style de la bordure, pour avoir une fenêtre non redimensionnable.
- La propriété `ShowInTaskBar` qui est positionnée à **False** pour que la fenêtre n'apparaisse pas sur la barre des tâches.
- Il faut également prévoir un bouton de validation et un bouton d'annulation pour la fermeture de la boîte de dialogue.

Vous pouvez créer une boîte de dialogue en modifiant manuellement ces propriétés sur une fenêtre normale, ou choisir au moment de l'ajout d'un élément au projet l'option 'Dialog'.



L'affichage de la boîte de dialogue se fera par l'appel de la méthode `ShowDialog` au lieu de la méthode `Show` car la méthode `ShowDialog` affiche la fenêtre en mode modal (notre boîte de dialogue sera la seule partie utilisable de l'application tant qu'elle sera ouverte).

À la fermeture de la boîte, il faut pouvoir déterminer quel bouton a provoqué la fermeture de la boîte de dialogue. C'est en fait la méthode `Showdialog` qui va nous fournir la solution. Elle nous retourne une des valeurs de l'énumération `System.Windows.Forms.DialogResult`. La valeur retournée n'est bien sur pas prise au hasard. Vous êtes donc obligés, au moment de la conception de la boîte de dialogue, de fournir la valeur à retourner pour chacun des boutons provoquant la fermeture de la boîte de dialogue. Vous pouvez le faire en modifiant la propriété `DialogResult` de la boîte de dialogue dans l'événement `Click` de chacun des boutons ou en modifiant la propriété `DialogResult` des boutons concernés par la fermeture de la boîte de dialogue. À noter que dans ce cas il n'y a pas besoin de gérer l'événement `Click` sur le bouton pour provoquer la fermeture de la boîte de dialogue. Si les deux solutions sont utilisées simultanément, la propriété `DialogResult` de la boîte de dialogue sera prioritaire pour déterminer la valeur renvoyée par la méthode `ShowDialog`.

Maintenant que nous savons comment configurer et afficher une boîte de dialogue, le plus dur reste à faire : créer l'interface visuelle de la boîte de dialogue.

Utilisation des contrôles

Les contrôles vont nous permettre de créer l'interface entre l'application et son utilisateur. C'est par leur intermédiaire que l'utilisateur pourra agir sur le fonctionnement de l'application en saisissant du texte, en choisissant des options, en lançant l'exécution d'une partie spécifique de notre application, etc.

Les contrôles seront disponibles, en VB.NET, par l'intermédiaire d'une série de classes qui devront être instanciées au cours de l'exécution de l'application.

Ces différentes classes sont issues d'une hiérarchie qui commence par la classe de base `Control`. Cette classe assure les fonctions élémentaires des contrôles (positions, dimensions...) puis une classe dérivée ajoute des fonctionnalités supplémentaires et ainsi de suite jusqu'à la classe finale de la hiérarchie.

1. Ajout de contrôles

Les contrôles peuvent être ajoutés sur une fenêtre de deux manières différentes. La plus simple, et la plus rapide également, passe par l'utilisation de la boîte à outils. Ici encore il existe trois possibilités pour l'ajout des contrôles.

- Effectuez un double clic sur le contrôle dans la boîte à outils. Cette méthode permet d'en placer un exemplaire, avec une taille par défaut au centre de la fenêtre.

- Effectuez un glisser-déplacer entre la boîte à outils et la fenêtre. Dès que vous survolez la feuille, le curseur de la souris vous indique, par la présence d'un petit signe plus (+), que vous allez ajouter quelque chose sur votre feuille. La position à laquelle vous lâcherez le bouton de votre souris correspondra à la position du coin supérieur gauche de votre contrôle. Il sera dimensionné avec les valeurs par défaut.

- Sélectionnez le contrôle dans la boîte à outils puis cliquer sur la fenêtre à l'endroit où vous voulez placer le coin supérieur gauche de votre contrôle, puis sans relâcher le bouton de la souris, agrandissez le rectangle jusqu'à la taille désirée pour votre contrôle.

Si vous souhaitez placer plusieurs exemplaires du même contrôle sur votre fenêtre, il est possible de bloquer la sélection dans la boîte à outils en utilisant la touche [Ctrl] lorsque vous sélectionnez le contrôle dans la boîte à outils. Vous pourrez alors placer plusieurs exemplaires du même contrôle sans avoir à le resélectionner dans la boîte à outils en conservant la touche [Ctrl] enfoncée.

Certains contrôles n'ont pas d'interface visible au moment de la conception de la fenêtre. Pour éviter d'encombrer la surface de la fenêtre, ils sont placés dans une zone située en bas de la fenêtre de conception graphique ; c'est le cas par exemple des contrôles `ImageList` et `Timer` que nous verrons un peu plus loin dans ce chapitre. Il est possible d'ajouter des contrôles à la boîte à outils. Ces contrôles peuvent être des contrôles .net ou des contrôles ActiveX. L'utilisation de contrôles ActiveX va entraîner quelques inconvénients pour votre application. Le code de votre application sera moins efficace (des opérations supplémentaires seront nécessaires pour accéder au contrôle ActiveX).

Le déploiement de votre application nécessitera des modifications dans la base de registre des machines pour l'enregistrement des contrôles ActiveX.

Les contrôles ajoutés sont nommés automatiquement par Visual Studio au fur et à mesure de l'ajout. Par contre, les noms utilisés par défaut ne sont pas très explicites.

Le code suivant ne doit pas vous paraître très limpide.

```
Button1.Enabled = False
TextBox1.Clear()
CheckBox1.Checked = True
RadioButton1.Checked = False
RadioButton2.Checked = True
```

Il est donc primordial, pour la lisibilité du code, de renommer les contrôles de préférence dès la création ou, au plus tard, avant de les utiliser dans le code. Il suffit simplement de changer la propriété `name` de chacun d'entre eux à l'aide de la fenêtre de propriétés. Il n'y a pas de règle absolue à respecter pour les noms des contrôles. Une solution fréquemment utilisée consiste à associer un préfixe représentatif du type du contrôle à un nom explicite pour l'application. Les préfixes ne sont pas normalisés mais les règles suivantes subsistent depuis les premières versions de Visual Basic.

Préfixe	Contrôle
cbo	ComboBox
lst	Listbox

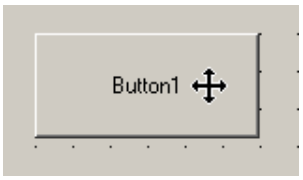
chk	CheckBox
opt	RadioButton
cmd	Button
txt	TextBox
lbl	Label

Avec le respect de ces conventions et un peu de bon sens, le code est nettement plus clair :

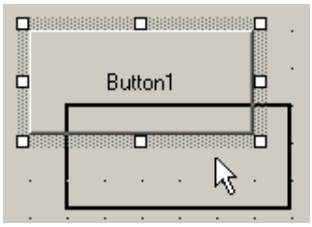
```
cmdValidation.Enabled = False
txtNom.Clear()
chkItalique.Checked = True
optBleu.Checked = False
optVert.Checked = True
```

2. Position et dimension des contrôles

Après avoir placé les contrôles sur la fenêtre, il est bien sûr possible de les repositionner ou de les redimensionner. Lorsque vous déplacez la souris au-dessus d'un contrôle, le curseur change d'apparence pour indiquer la possibilité de déplacer le contrôle.



Il suffit de cliquer sur le contrôle puis de déplacer la souris. Un rectangle noir suit alors votre curseur de souris, pour représenter la future position de votre contrôle. Des lignes de guidage sont affichées pendant le déplacement du contrôle pour faciliter son alignement avec les autres contrôles déjà placés sur la fenêtre. Les lignes bleues représentent les alignements possibles sur les bordures des autres contrôles, les lignes roses représentent les alignements possibles sur les libellés des contrôles. Le contrôle sera effectivement déplacé au moment où vous relâcherez le bouton de votre souris.



L'utilisation des flèches du clavier est également possible et apporte plus de précision dans les déplacements.

Vous pouvez aussi modifier la position d'un contrôle par sa propriété `Location` dans la fenêtre de propriétés. Cette propriété est d'ailleurs modifiée lorsque vous déplacez un contrôle avec la souris ou le clavier.

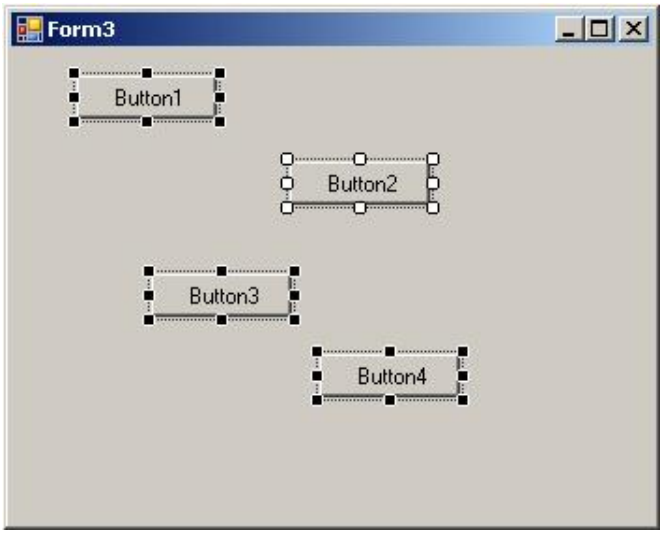
Enfin, la dernière possibilité est de modifier les propriétés `Left` et `Top` du contrôle par le code. Le morceau de code suivant permet de déplacer le bouton de commande à une position aléatoire à chaque fois que vous cliquez dessus.

```
Private Sub cmdtest_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) Handles cmdtest.Click
        cmdtest.Left = Rnd() * (Me.ClientSize.Width - cmdtest.Size.Width)
        cmdtest.Top = Rnd() * (Me.ClientSize.Height - cmdtest.Size.Height)
End Sub
```

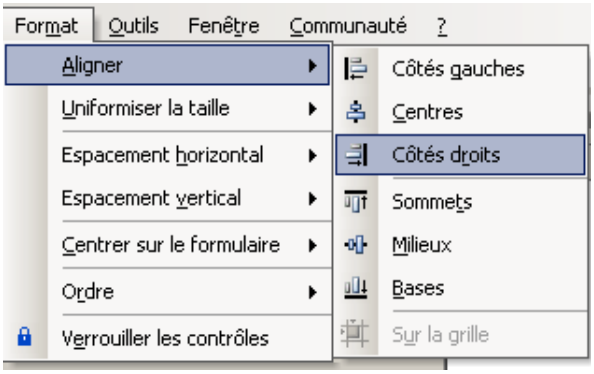
Des fonctionnalités plus évoluées permettent le positionnement des contrôles les uns par rapport aux autres. Pour pouvoir les utiliser, il faut au préalable sélectionner plusieurs contrôles sur votre feuille. Pour cela, deux solutions possibles :

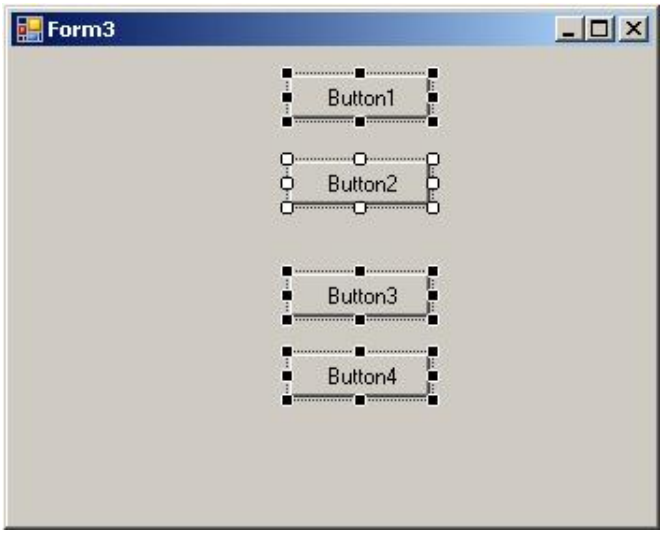
- Dessiner un rectangle de sélection, avec la souris, autour des contrôles.

- Cliquer sur les contrôles les uns après les autres en maintenant la touche [Ctrl] enfoncée. Le premier contrôle sélectionné apparaît avec des poignées de sélection blanches.



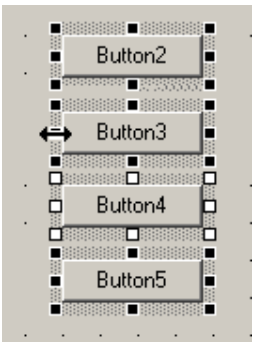
Les options du menu **Format** de VB.NET sont maintenant disponibles et vous fournissent de nombreuses options pour positionner les contrôles. Le contrôle qui apparaît dans la sélection avec des poignées de sélection blanches est considéré comme référence pour l'alignement.





De nombreuses autres options sont disponibles pour organiser le placement des contrôles sur votre feuille.

Le redimensionnement des contrôles est également très simple à mettre en œuvre puisqu'il suffit de sélectionner le ou les contrôles à redimensionner et de placer le curseur de la souris sur l'une des poignées de sélection, pour faire apparaître une flèche vous indiquant dans quelle direction vous pouvez redimensionner le contrôle. Il faut alors cliquer sur le carré correspondant et déplacer la souris jusqu'à ce que le contrôle ait atteint la taille désirée.



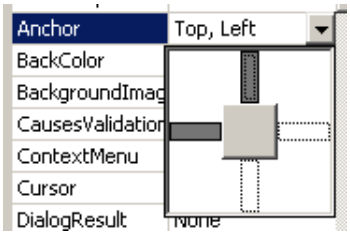
Vous pouvez également utiliser les flèches du clavier en association avec la touche [Shift] pour le dimensionnement des contrôles.

Le redimensionnement par le code utilise la méthode `SetBounds` qui permet à la fois de fixer la position et la taille du contrôle. Le code suivant diminue la taille du bouton à chaque fois que l'on clique dessus.

```
Private Sub cmdMincir_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles cmdMincir.Click
cmdMincir.SetBounds(cmdMincir.Left, cmdMincir.Top, cmdMincir.Size.Width - 5,
cmdMincir.Size.Height - 5)
End Sub
```

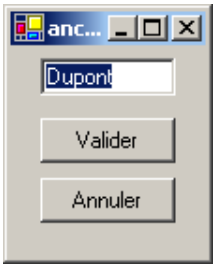
Après de nombreux efforts pour positionner et dimensionner les contrôles, il serait dommage qu'une erreur de manipulation vienne remettre tout en cause. Pour éviter ces soucis, il est possible de verrouiller les contrôles sur la feuille, par le menu **Format - Verrouiller les contrôles**. Cette commande bloque le déplacement et le redimensionnement de tous les contrôles présents sur la feuille ainsi que le redimensionnement de la feuille elle-même. Les contrôles peuvent ensuite être déverrouillés par la même option de menu. Vous pouvez également déverrouiller les contrôles individuellement par la propriété `Locked`.

Si vous concevez une application dans laquelle l'utilisateur peut redimensionner la fenêtre au moment de l'exécution, les contrôles doivent suivre les modifications de taille de la fenêtre. Pour autoriser le redimensionnement automatique d'un contrôle, vous pouvez utiliser la propriété `Anchor` du contrôle. Par l'intermédiaire de cette propriété, vous indiquez que la distance entre les bords du contrôle et les positions d'ancrage sera conservée lors du redimensionnement de la fenêtre. À la création, les contrôles sont ancrés aux bords haut et gauche de la feuille. La modification de cette propriété s'effectue par un petit assistant, disponible dans la fenêtre de propriétés.



Pour modifier la propriété `anchor`, sélectionnez la branche de l'étoile correspondant au côté avec lequel vous voulez réaliser un ancrage, ou supprimer un ancrage existant.

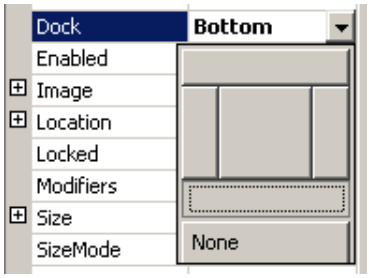
Par exemple, pour la fenêtre suivante, les contrôles sont ancrés à gauche et à droite.



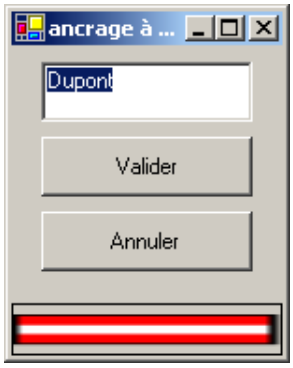
Si nous redimensionnons la fenêtre, les contrôles suivent l'agrandissement horizontal de la feuille.

Vous pouvez également indiquer qu'un contrôle doit adapter une ou plusieurs de ces dimensions à celle de son conteneur. Pour cela, vous utilisez la propriété`Dock` du contrôle en indiquant sur quelle bordure de son conteneur notre contrôle va adapter une de ses dimensions.

Par exemple, nous pouvons placer un contrôle `PictureBox` en demandant à l'amarrer à la bordure basse de la fenêtre.



Notre PictureBox s'adapte automatiquement à la largeur de la fenêtre et reste collée à sa bordure basse.



3. Passage du focus entre contrôles

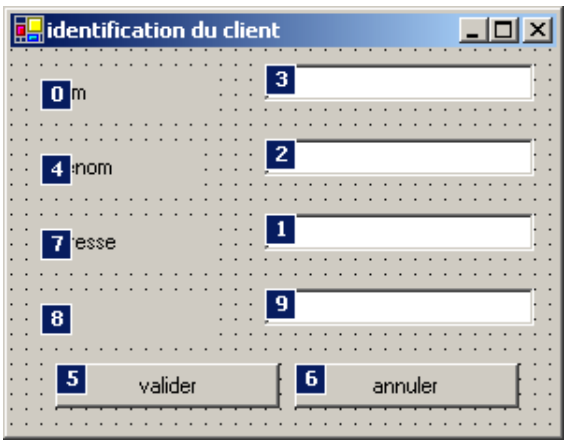
Lorsque vous concevez votre application, vous devez penser aux personnes réfractaires à l'utilisation de la souris en leur permettant quand même d'utiliser l'application. Il convient donc de concevoir l'application pour qu'elle puisse fonctionner uniquement en utilisant le clavier (sans clavier ni souris ce sera beaucoup plus difficile !).

Dans une application Windows, on dit d'un contrôle qu'il détient le focus lorsqu'il est prêt à recevoir la saisie de l'utilisateur. Le focus peut se déplacer de contrôle en contrôle en utilisant la touche [Tab]. Deux propriétés des contrôles règlent le passage du focus par la touche [Tab].

- La propriété `TabStop` indique si un contrôle pourra recevoir le focus par l'utilisation de la touche [Tab].
- La propriété `TabIndex` indique l'ordre dans lequel le focus sera passé entre les contrôles.

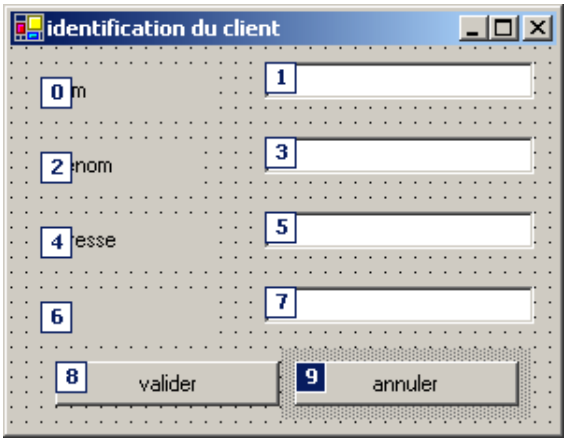
Par défaut, les propriétés `TabIndex` sont numérotées dans l'ordre dans lequel les contrôles sont créés.

Pour modifier cet ordre, vous pouvez modifier directement la propriété `TabIndex` de chaque contrôle ou utiliser le menu **Affichage - Ordre de tabulation**. Les contrôles sont alors affichés avec, dans leur coin supérieur gauche, la valeur de leur propriété `TabIndex`.



Vous devez ensuite cliquer sur les contrôles, dans l'ordre dans lequel vous voulez que le focus soit passé.

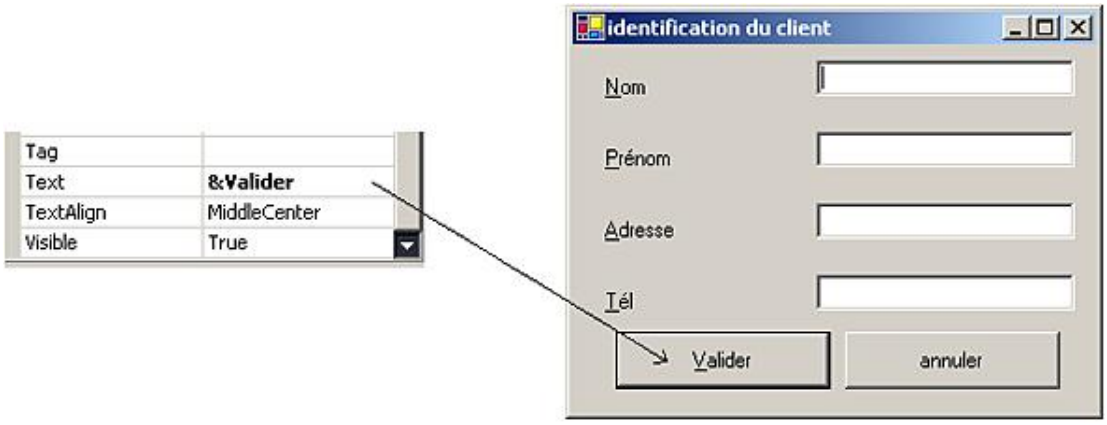
L'ordre suivant semble beaucoup plus logique pour cette boîte de dialogue.



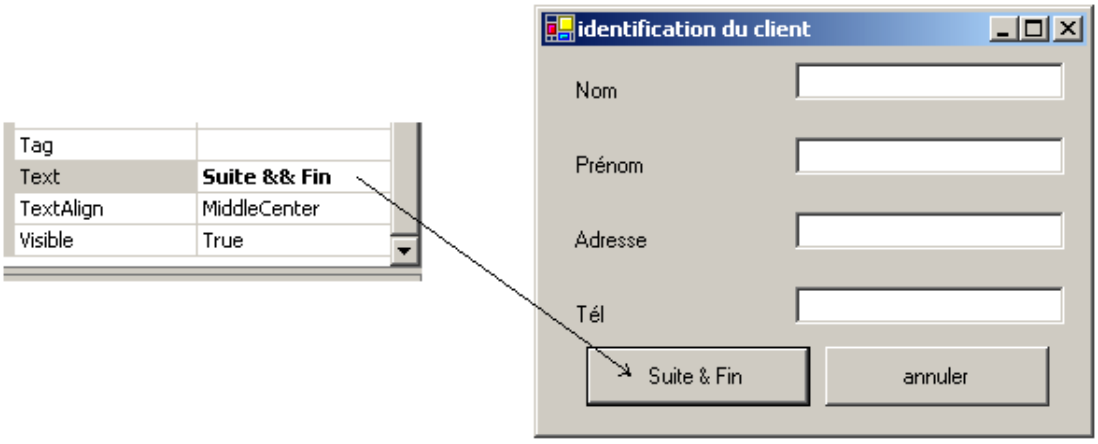
Vous pouvez ensuite revenir en mode normal en utilisant à nouveau le menu **Affichage - Ordre de tabulation** ou en utilisant la touche [Echap].

4. Raccourcis-clavier

Certains utilisateurs pressés souhaitent pouvoir se déplacer directement sur un contrôle particulier sans avoir à passer le focus sur tous ceux qui le précèdent dans l'ordre des tabulations. Vous pouvez, pour cela, ajouter un raccourci-clavier qui sera activé par l'intermédiaire de la touche [Alt] et un caractère. Pour spécifier le caractère à utiliser pour activer le contrôle, il faut ajouter dans la propriété `Text` du contrôle un caractère & devant le caractère utilisé pour le raccourci-clavier associé au contrôle. Cela provoque l'activation du raccourci et le soulignement du caractère dans le texte apparaissant sur le contrôle.



Si par contre, vous voulez insérer un caractère & dans la légende de votre contrôle, il faut le répéter deux fois dans sa propriété `Text`.



Pour certains contrôles (boutons, case à cocher, bouton d'option...), l'utilisation du raccourci-clavier est équivalent à un clic de souris et lance l'action correspondante, pour les autres, le raccourci-clavier place simplement le focus sur le contrôle correspondant.

Pour les contrôles n'ayant pas de légende, il faudra passer par l'intermédiaire d'un contrôle `label` qui lui servira de légende et activera également le raccourci-clavier. Nous verrons cela un peu plus loin dans ce chapitre.

Maintenant que nous savons utiliser les contrôles dans une application, nous allons examiner dans le détail les plus utilisés.

Les contrôles de VB.NET

Chaque contrôle utilisable dans VB.NET est représenté par une classe dont nous allons pouvoir créer des instances pour concevoir l'interface de l'application. La majorité des contrôles dérivent de la classe `Control` et de ce fait, héritent d'un bon nombre de propriétés de méthodes et d'événements.

Nous allons donc étudier les éléments les plus utiles de la classe `Control`.

1. La classe Control

a. Dimensions et position

Les propriétés `Left`, `Top`, `Width`, `Height` permettent le positionnement des contrôles. Ces propriétés peuvent être modifiées individuellement et acceptent des valeurs de type `Integer`.

Il est donc possible dans notre code d'utiliser la syntaxe suivante :

```
TextBoxNom.Left = 100
TextBoxNom.Top = 50
TextBoxNom.Width = 150
TextBoxNom.Height = 50
```

Deux autres propriétés permettent de travailler avec la position et la taille d'un contrôle : la propriété `Location` accepte un objet de type point grâce auquel nous pouvons spécifier la position de notre contrôle ; de la même manière, la propriété `Size` qui accepte un objet de type `Size`, gère les dimensions du contrôle. Les lignes précédentes peuvent être remplacées par :

```
TextBoxNom.Location = New Point(100, 50
TextBoxNom.Size = New Size(150, 50)
```

dans lesquelles nous construisons une instance de `Point` et de `Size`, avant de les associer aux propriétés correspondantes.

Une troisième possibilité nous permet de manipuler, à la fois, la position et la taille des contrôles : la propriété `Bounds` attend une instance de la classe `Rectangle` pour définir les caractéristiques du contrôle. Notre code se résume donc à une seule ligne :

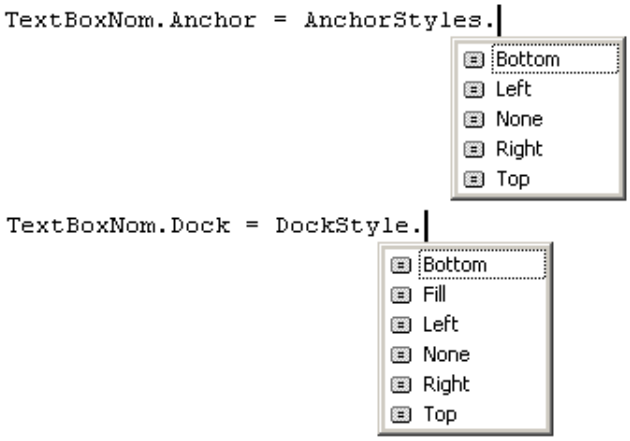
```
TextBoxNom.Bounds = New Rectangle(100, 50, 150, 50)
```

La méthode `SetBounds` permet également de modifier les positions et dimensions des contrôles sans avoir à créer de nouvelle instance de la classe `Rectangle` mais en modifiant celle déjà associée au contrôle.

```
TextBoxNom.SetBounds(100, 50, 150, 50)
```

La modification de ces propriétés entraîne le déclenchement des événements `Resize` et `Move` sur le contrôle. Ces événements sont bien sûr déclenchés lorsque la valeur des propriétés est modifiée dans le code, mais aussi lorsque, par exemple, la modification de taille de la fenêtre entraîne un repositionnement ou un redimensionnement du contrôle.

Le comportement des contrôles lorsque la fenêtre est redimensionnée est spécifié par les propriétés `Anchor` et `Dock`. Nous avons déjà vu comment modifier ces propriétés, par l'intermédiaire de la fenêtre de propriétés. Pour les modifier par le code, il suffit d'y affecter une des valeurs définies dans les énumérations `AnchorStyle` et `DockStyle`.



Jusqu'à présent, les positions avec lesquelles nous avons travaillé étaient des positions exprimées par rapport au coin supérieur gauche du conteneur du contrôle. Dans certains cas, il peut être utile d'obtenir les coordonnées d'un point du contrôle non pas rapport au coin supérieur gauche du contrôle, mais par rapport au coin supérieur gauche de l'écran. La méthode `PointToScreen` permet cette conversion. Elle attend, comme paramètre, une instance de la classe `Point` avec les coordonnées exprimées par rapport au contrôle et renvoie une nouvelle instance de la classe `Point` avec les coordonnées exprimées par rapport à l'écran.

Le code suivant convertit, en coordonnées écran, la position supérieure gauche d'un contrôle TextBox :

```
        System.Console.Write("Controle/fenetre :")
        System.Console.WriteLine(Button2.Location)
        Dim p As Point
        p = Button2.PointToScreen(Button2.Location)
        System.Console.Write("Controle/ecran :")
        System.Console.WriteLine(p)
```

Résultat :

```
Controle/fenetre :{X=107,Y=72}
Controle/ecran :{X=306,Y=255}
```

L'opération inverse peut être réalisée par la méthode `pointToClient` qui elle, prend comme paramètre un point en coordonnées écran et renvoie un point exprimé en coordonnées liées au contrôle. Si l'on effectue l'opération inverse, c'est-à-dire en partant des coordonnées écran, on obtient bien la même valeur :

```
System.Console.WriteLine("controle / fenetre à partir de l'écran" &
Button2.PointToClient(p).ToString)
```

Résultat

```
controle / fenetre à partir de l'écran{X=107,Y=72}
```

b. Apparence des contrôles

La couleur de fond du contrôle peut être modifiée par la propriété `BackColor` tandis que la couleur du texte du contrôle est modifiée par la propriété `ForeColor`.

On peut affecter à ces propriétés des valeurs définies dans l'espace de nom `System.Drawing.Color` pour obtenir des couleurs prédéfinies dans Visual Basic.

```
TextBoxNom.BackColor = System.Drawing.Color.Yellow
```

On peut également utiliser les constantes définies dans l'espace de nom `System.Drawing.SystemColors` pour utiliser une des couleurs définies au niveau du système lui-même. L'intérêt, dans ce cas, est que votre application va s'adapter en fonction de la configuration de la machine sur laquelle elle fonctionnera.

```
TextBoxNom.BackColor=System.Drawing.SystemColors.InactiveCaptionText
```

La troisième solution consiste à effectuer le mélange de couleur vous-même, en utilisant la fonction `FromArgb` et en spécifiant comme paramètre la quantité de chacune des couleurs de base (Rouge, Vert, Bleu).

```
TextBoxNom.BackColor = System.Drawing.Color.FromArgb (127, 0, 127)
```

La police est modifiable par la propriété `Font` du contrôle. On peut, pour l'occasion, créer une nouvelle instance de la classe `Font` et l'affecter au contrôle. Il y a treize constructeurs différents pour la classe `Font` donc treize façons différentes de créer une police de caractère. Nous utiliserons la plus simple en indiquant, simplement, le type de police et la taille.

```
TextBoxNom.Font = New Font(System.Drawing.FontFamily.GenericMonospace, 16)
```

Après avoir effectué des modifications sur ces propriétés, il est possible de revenir à une configuration normale en appelant les méthodes `ResetBackColor`, `ResetForeColor`, `ResetFont`. Les propriétés correspondantes sont réinitialisées avec les valeurs définies pour le conteneur du contrôle.

La propriété `BackgroundImage` permet de spécifier une image qui sera utilisée comme fond pour le contrôle. Si l'image n'est pas assez grande pour recouvrir le contrôle, elle est représentée en mosaïque :

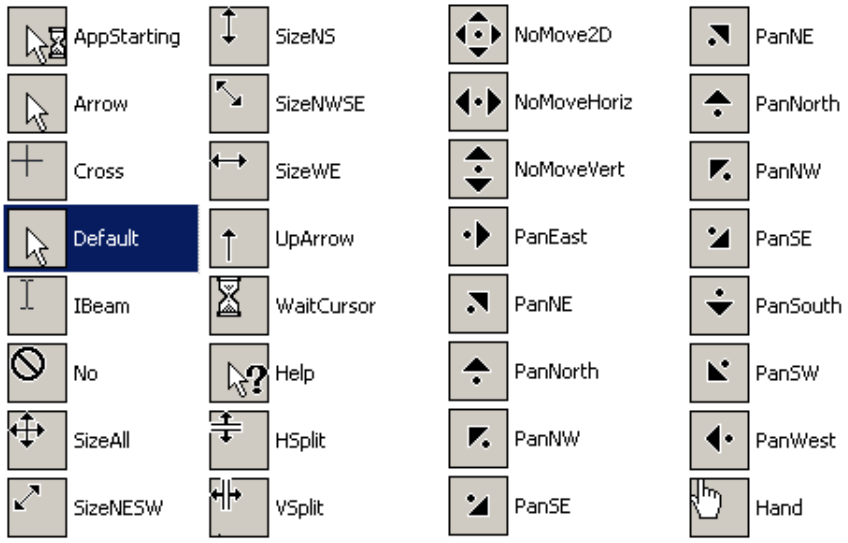
```
BtnValider.BackgroundImage = New Bitmap("open.bmp")
```

Le résultat est parfois déroutant !

Pour revenir à quelque chose de plus classique :

```
BtnValider.BackgroundImage = Nothing
```

La propriété `Cursor` permet de choisir l'apparence du curseur lorsque la souris se trouve sur la surface du contrôle. Plusieurs curseurs sont prédéfinis dans Windows.



Ces curseurs sont rangés dans une collection **Cursors** et peuvent être utilisés directement en les affectant à la propriété `Cursor` du contrôle.

```
BtnValider.Cursor = Cursors.WaitCursor
```

Si, parmi ceux-ci, aucun ne vous convient, vous pouvez utiliser un curseur personnalisé en créant une instance de la classe `Cursor` et en l'affectant à la propriété `Cursor` du contrôle.

```
BtnValider.Cursor = New Cursor("h_nodrop.cur")
```

La détection de l'entrée et de la sortie de la souris sur le contrôle et la modification du curseur en conséquence est gérée automatiquement par le contrôle lui-même.

Comme pour la police de caractère, il est possible de restaurer le curseur par défaut en appelant la méthode `ResetCursor`.

La modification de la plupart des propriétés des contrôles déclenche un événement. Ces événements sont identifiés

par le nom de la propriété suivi du suffixe Changed. Ils peuvent être utilisés pour sauvegarder les préférences de l'utilisateur lorsqu'il personnalise l'application.

```
Private Sub Form1_BackColorChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.BackColorChanged
        My.Settings.CouleurFond = Me.BackColor
        My.Settings.Save()
End Sub
```

c. Comportement des contrôles

Les contrôles placés sur une feuille peuvent être masqués en modifiant la propriété Visible ou désactivés en modifiant la propriété Enabled. Dans ce cas, le contrôle est toujours visible mais apparaît avec un aspect grisé pour indiquer à l'utilisateur que ce contrôle est inactif pour le moment.

```
BtnValider.Enabled = False
```

Les contrôles dans cet état ne peuvent bien sûr pas recevoir le focus dans l'application. Vous pouvez vérifier cela en examinant la propriété CanFocus qui renvoie un Boolean. Vous pouvez également vérifier si un contrôle détient actuellement le focus, en vérifiant la propriété Focused ou la propriété ContainsFocus. Cette dernière est à utiliser avec les contrôles conteneurs (c'est-à-dire les contrôles qui peuvent contenir d'autres contrôles), dans ce cas cette propriété est positionnée sur **True** si l'un des contrôles placés à l'intérieur du conteneur a le focus.

Le focus peut être placé sur un contrôle sans l'intervention de l'utilisateur, en appelant la méthode focus du contrôle.

```
BtnValider.Focus()
```

Pour surveiller le passage du focus d'un contrôle à l'autre, quatre événements sont à votre disposition :

- Enter indique que le focus arrive sur un des contrôles d'un conteneur.
- GotFocus indique qu'un contrôle particulier a reçu le focus.
- LostFocus indique qu'un contrôle a perdu le focus.
- Leave indique que le focus n'est plus sur un des contrôles du conteneur.

Par exemple, pour bien visualiser qu'un contrôle a le focus, on peut utiliser le code suivant qui modifie la couleur du texte lorsque le contrôle reçoit ou perd le focus :

```
Private Sub TxtNom_GotFocus(ByVal sender As Object, ByVal e As System.
EventArgs) Handles TextBoxNom.GotFocus
        TextBoxNom.ForeColor = System.Drawing.Color.Red
    End Sub

    Private Sub TxtNom_LostFocus(ByVal sender As Object, ByVal e As System.
EventArgs) Handles TextBoxNom.LostFocus
        TextBoxNom.ResetForeColor()
    End Sub
```

Dans certains cas, il est souhaitable de vérifier la saisie de l'utilisateur dans un formulaire avant de continuer dans l'application. Cette vérification peut être effectuée à la fermeture du formulaire ou au fur et à mesure de la saisie, dans les différents contrôles du formulaire.

Chaque contrôle peut être configuré pour permettre la vérification de la saisie en modifiant la propriété CausesValidation sur **True**. Juste avant que le contrôle ne perde le focus, l'événementValidating est déclenché pour permettre la vérification de la saisie de l'utilisateur. Si la saisie n'est pas correcte (en fonction des critères que nous avons fixés), nous pouvons bloquer le passage du focus vers un autre contrôle en modifiant la propriété Cancel de l'objetCancelEventArg qui est passé comme paramètre. Dans ce cas, le focus reste sur le contrôle pour lequel la saisie n'est pas correcte. Par contre, si la saisie est correcte, l'événementValidated est déclenché sur le contrôle et le focus se déplace sur le contrôle suivant.

Par exemple, pour saisir un numéro de téléphone, nous pouvons vérifier que seules des valeurs numériques ont été saisies. En cas d'erreur nous générons un beep, modifions la couleur du texte et bloquons le passage du focus sur un autre contrôle.

```
Private Sub TxtTel_Validating(ByVal sender As Object, ByVal e As System.
ComponentModel.CancelEventArgs) Handles txtTel.Validating
If Not IsNumeric(txtTel.Text) And txtTel.Text  "" Then
            Beep()
            txtTel.ForeColor = System.Drawing.Color.Red
            e.Cancel = True
        End If
    End Sub

    Private Sub TxtTel_Validated(ByVal sender As Object, ByVal e As System.
EventArgs) Handles txtTel.Validated
        txtTel.ResetForeColor()
    End Sub
```

Deux propriétés sont parfois utiles lorsque nous travaillons avec des contrôles conteneur. La propriété `HasChildren` nous permet de savoir si des contrôles sont placés dans notre conteneur. Si c'est le cas, la collection `Controls` contient la liste de tous ces contrôles. Nous pouvons par exemple modifier la couleur de texte de tous les contrôles d'un conteneur lorsque le focus est placé sur l'un d'entre eux.

```
Private Sub GroupBox1_Enter(ByVal sender As Object, ByVal e As
System.EventArgs) Handles GrBoxIdent.Enter
        If GrBoxIdent.HasChildren Then
            Dim x As Object
            For Each x In GrBoxIdent.Controls
                x.forecolor = System.Drawing.Color.YellowGreen
            Next
        End If
    End Sub

    Private Sub GroupBox1_Leave(ByVal sender As Object, ByVal e As System.
EventArgs) Handles GrBoxIdent.Leave
        If GrBoxIdent.HasChildren Then
            Dim x As Object
            For Each x In GrBoxIdent.Controls
                x.resetforecolor()
            Next
        End If
    End Sub
```

L'opération inverse est également possible c'est-à-dire, qu'à partir d'un contrôle, nous pouvons récupérer les propriétés de son conteneur. La propriété `Parent` fournit une référence vers le conteneur du contrôle. Nous pouvons, par exemple, faire en sorte que la couleur de fond de chaque contrôle change en même temps que celle de son conteneur.

```
Private Sub GrBoxIdent_BackColorChanged(ByVal sender As Object, ByVal e As
System.EventArgs) Handles GrBoxIdent.BackColorChanged
        If GrBoxIdent.HasChildren Then
            Dim x As Object
            For Each x In GrBoxIdent.Controls
                x.backcolor = x.parent.backcolor
            Next
        End If
    End Sub
```

Maintenant que nous avons exploré les propriétés communes aux différents contrôles disponibles, nous allons les étudier un par un en explorant leurs spécificités.

2. Les contrôles d'affichage d'informations

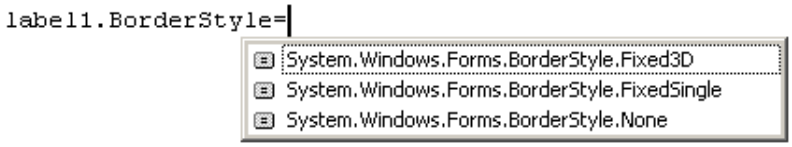
a. Le contrôle Label

Le contrôle Label est utilisé pour afficher, sur un formulaire, un texte qui ne sera pas modifiable par l'utilisateur. Il sert

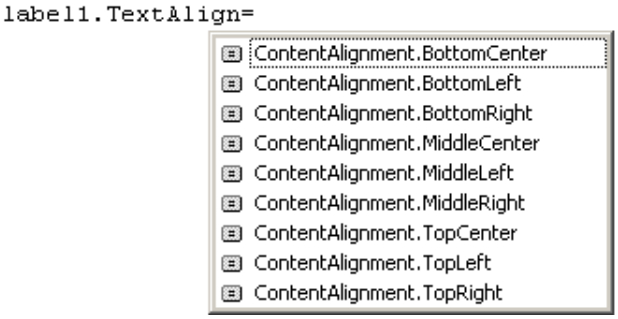
essentiellement à fournir une légende à des contrôles qui n'en possède pas (zones de texte par exemple, liste déroulante...). Dans ce cas, il permettra également de fournir un raccourci-clavier pour atteindre le contrôle.

Le texte affiché par le contrôle est indiqué par la propriété `Text`. Cette propriété pourra bien sûr être modifiée par le code l'application. Il faut cependant être prudent car, par défaut, le contrôle conservera la taille que vous lui avez donnée à la conception. Si la nouvelle chaîne de caractères affectée à la propriété `Text` est plus longue que celle spécifiée au moment de la conception, seul le début sera visible. Pour éviter ce problème, il faut demander au contrôle `Label` d'adapter sa largeur en fonction du texte à afficher, en modifiant la propriété `AutoSize` sur **True**.

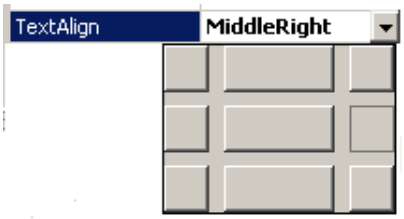
Par défaut, le contrôle `Label` n'a pas de bordure. Vous pouvez en ajouter une en modifiant la propriété `BorderStyle`, en utilisant l'une des trois valeurs disponibles.



Vous avez aussi la possibilité d'indiquer la position du texte dans le contrôle par l'intermédiaire de la propriété `TextAlign`. Dans le code, vous utiliserez l'une des constantes prédéfinies.



Par la fenêtre de propriété, il suffit de cliquer sur la position désirée pour le texte à l'intérieur de votre contrôle.



À noter cependant que la propriété `TextAlign` modifiera la position du texte uniquement si la propriété `AutoSize` est positionnée sur **False**.

Les contrôles `Label` peuvent également afficher des images. Vous pouvez indiquer l'image à afficher à l'aide de la propriété `Image`. Une autre solution consiste à utiliser un contrôle `ImageList` qui servira, en quelque sorte, de stockage pour les images de l'application. Dans ce cas, vous indiquez, par l'intermédiaire de la propriété `ImageList`, dans quel contrôle vous allez chercher l'image, et par la propriété `ImageIndex` à quel endroit elle se trouve dans le contrôle `ImageList`. Si vous utilisez un contrôle `ImageList`, la propriété `Image` de votre contrôle sera ignorée. Comme pour le texte, vous pouvez modifier la position de l'image dans le contrôle par la propriété `ImageAlign` avec les mêmes constantes que pour la propriété `TextAlign`.

Nous avons indiqué que le contrôle `Label` pouvait être utilisé comme raccourci-clavier pour un autre contrôle. Pour cela, trois précautions sont à prendre.

- Comme pour les autres contrôles, ajouter un & dans la propriété `Text` pour le caractère utilisé comme raccourci.
- Indiquer au contrôle `Label` son rôle de gestionnaire de raccourci-clavier en modifiant la propriété `UseMnemonic` sur **True**.
- Vérifier que le contrôle, qui doit recevoir le focus, est immédiatement après le contrôle `Label` dans l'ordre des tabulations (propriété `TabIndex`).

b. Le contrôle LinkLabel

Le contrôle `LinkLabel` hérite de toutes les caractéristiques du contrôle `Label` et ajoute simplement des fonctionnalités de lien style `Web`. Les propriétés supplémentaires par rapport au contrôle `Label` gèrent les différents paramètres du lien.

La propriété `LinkArea` indique quelle portion du texte activera le lien. Cette propriété peut être modifiée, par l'intermédiaire de la fenêtre de propriétés, avec un petit utilitaire dans lequel vous devez sélectionner la portion de texte constituant le lien.

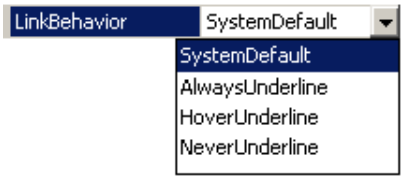
Les couleurs utilisées pour le lien sont modifiables par trois propriétés :

`LinkColor`
couleur du lien à l'état normal.

`VisitedLinkColor`
couleur du lien après une première utilisation.

`ActiveLinkColor`
couleur du lien au moment où l'on clique dessus.

L'apparence du lien est modifiable par la propriété `LinkBehavior`.



Les quatre valeurs possibles permettent respectivement :

- d'utiliser la même configuration, pour les liens, que votre navigateur ;
- d'avoir les liens toujours soulignés ;
- d'avoir les liens soulignés lorsque la souris les survole ;
- de ne jamais avoir les liens soulignés.

Lorsque l'utilisateur clique sur le lien, l'événement `LinkClicked` est déclenché dans l'application ; à vous d'écrire du code, pour exécuter une action dans votre application.

Vous devez également modifier la propriété `LinkVisited` en la positionnant sur **True**, pour indiquer que ce lien a déjà été utilisé dans l'application.

L'action peut être l'ouverture d'une page d'un site Web dans le navigateur par défaut, comme dans l'exemple suivant :

```
Private Sub LienMicrosoft_LinkClicked(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.LinkLabelLinkClickedEventArgs) Handles LinkLabel1.
```

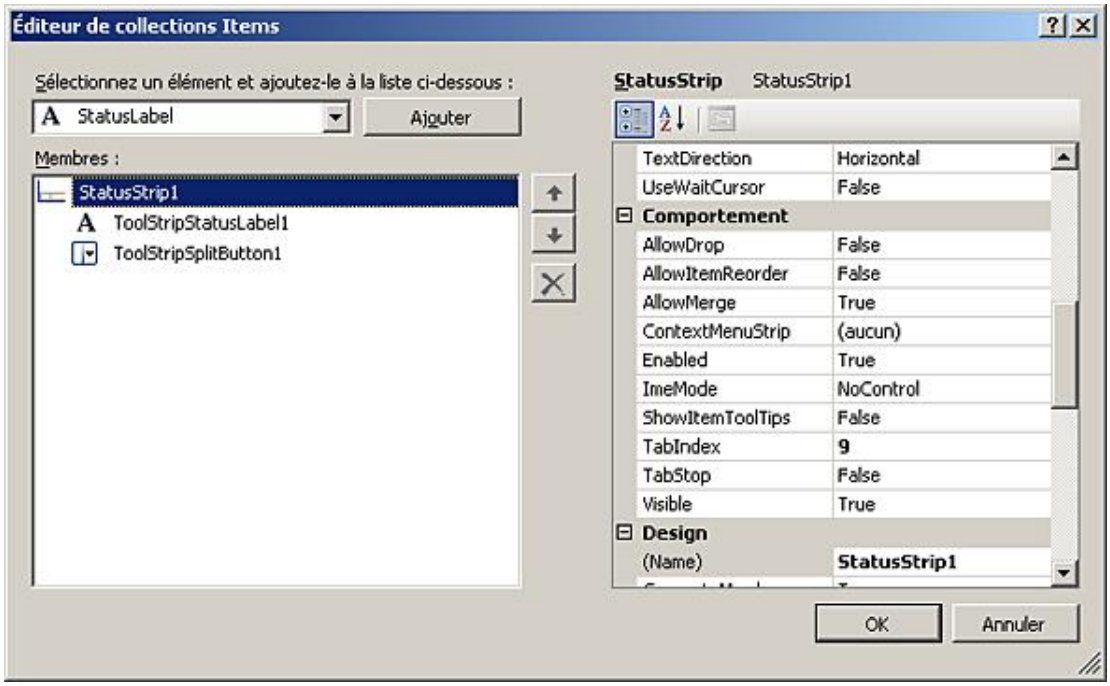
```
LinkClicked
     System.Diagnostics.Process.Start(" http ://www.microsoft.fr ")
     LinkLabel1.LinkVisited = True
End Sub
```

Ou encore, l'affichage d'une nouvelle feuille dans notre application comme dans l'exemple suivant :

```
Private Sub LienMicrosoft_LinkClicked(ByVal sender As System.Object, ByVal e
As System.Windows.Forms.LinkLabelLinkClickedEventArgs) Handles LienMicrosoft.
LinkClicked
     Dim feuille As dlgPerso
     feuille = New dlgPerso()
     feuille.ShowDialog()
     LienMicrosoft.LinkVisited = True
End Sub
```

c. Le controle StatusStrip

Le contrôle `StatusStrip` est généralement utilisé pour présenter des informations à l'utilisateur, concernant le fonctionnement de l'application. Il peut afficher les informations sur plusieurs types zones. Les informations peuvent être affichées sous forme de texte, de barre de progression, de menu ou de bouton de commande associé à un menu. Un éditeur spécifique accessible par la propriété `Items` du contrôle permet sa configuration.

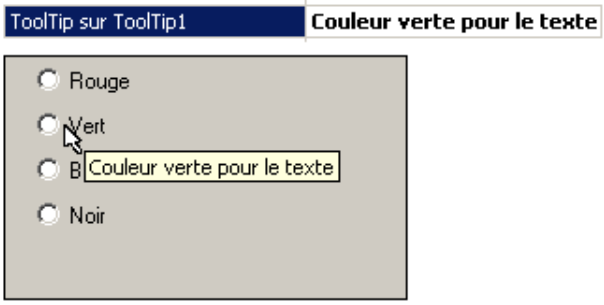


Chaque élément ajouté sur le contrôle `StatusStrip` doit ensuite être configuré individuellement. Les propriétés des éléments pouvant être utilisés pour la construction d'une `StatusStrip` sont très semblables à celles des contrôles normaux. Par exemple, l'élément `ToolStripStatusLabel` est pratiquement identique au contrôle `LinkLabel`.

d. Le contrôle ToolTip

Ce contrôle permet l'affichage d'une bulle d'aide associée à un contrôle. Ce contrôle n'a pas d'interface visible, il sera donc placé dans la zone située en dessous de la fenêtre de conception. Il effectue beaucoup de travail sans aucun effort de programmation. Il surveille, par exemple, en permanence où se trouve la souris ; si celle-ci est sur un contrôle, il vérifie s'il y a une info-bulle associée au contrôle et si c'est le cas, il affiche l'info-bulle pendant la durée spécifiée par la propriété `AutoPopDelay`.

Pour pouvoir fonctionner, le contrôle `ToolTip` doit associer une chaîne de caractères à chacun des contrôles de l'interface. Pour cela, dès qu'un contrôle `ToolTip` est disponible sur une feuille, une propriété `ToolTip` est ajoutée à chacun des contrôles, permettant ainsi de spécifier le texte de l'info-bulle associée au contrôle.



Les chaînes de caractères associées à chaque contrôle peuvent également être indiquées par l'intermédiaire du code, en appelant la méthode `SetToolTip` et en indiquant comme paramètre, le nom du contrôle et la chaîne de caractères, qui lui est associée.

```
ToolTip1.SetToolTip(RadioButton3, " Couleur bleu pour le texte ")
```

Cette technique permet de conserver des légendes relativement courtes pour les contrôles, tout en fournissant suffisamment d'informations sur l'utilisation de l'application.

e. Le Contrôle ErrorProvider

Ce contrôle permet d'indiquer facilement à l'utilisateur des problèmes sur les données qu'il a saisies, sur un formulaire. En général, il intervient lors de la phase de validation des données du formulaire, en affichant en face de chaque contrôle une petite icône  afin d'attirer l'attention de l'utilisateur. Des informations supplémentaires peuvent être fournies par une info-bulle associée au contrôle `ErrorProvider`.

Un même contrôle `ErrorProvider` peut être utilisé pour tous les contrôles d'un formulaire.

L'activation du contrôle `ErrorProvider` peut s'effectuer à la fermeture du formulaire lorsque l'utilisateur clique sur le bouton **OK**, mais il est également possible de surveiller la saisie au fur et à mesure où elle est effectuée en gérant par exemple les événements `Validating`. Cet événement est déclenché par un contrôle au moment où celui-ci perd le focus. Nous pouvons, dans ce cas, vérifier immédiatement la valeur saisie dans le contrôle et réagir en conséquence en affichant notre contrôle `ErrorProvider`. Pour cela, nous appelons la méthode `SetError` en spécifiant le nom du contrôle qui nous pose problème et la chaîne de caractères affichée dans l'info-bulle associée au contrôle. S'il n'y a pas d'erreur, il faut réinitialiser la chaîne pour faire disparaître l'icône du contrôle `ErrorProvider`.

```
Private Sub TextBox1_Validating(ByVal sender As Object, ByVal e As System.
ComponentModel.CancelEventArgs) Handles TextBox1.Validating
        If Not IsNumeric(TextBox1.Text) Then
          ErrorProvider1.SetError(TextBox1, " Valeur numérique Obligatoire ")
        Else
          ErrorProvider1.SetError(TextBox1, "")
        End If
End Sub
```

f. Le contrôle NotifyIcon

Ce contrôle est principalement utilisé pour afficher des informations sur le fonctionnement d'un processus s'exécutant en tâche de fond dans l'application. Il est affiché dans la zone de statut du système. La propriété `Icon` du contrôle détermine l'icône affichée. La propriété `Text` représente la légende affichée lorsque la souris survole le contrôle.

En gérant l'événement `DoubleClick` du contrôle, vous pouvez afficher une boîte de dialogue permettant la configuration du processus associé au contrôle.

```
Private Sub IconService_DoubleClick(ByVal sender As Object, ByVal e As
System.EventArgs) Handles IconService.DoubleClick
        Dim dlg As ConfigService
        dlg = New ConfigService
        dlg.ShowDialog()
    End Sub
```

Il est également possible d'associer un menu contextuel, en renseignant la propriété `ContextMenuStrip`. Ce menu peut, par exemple, contrôler le fonctionnement du processus auquel est associé le contrôle.

g. Le contrôle HelpProvider

Le contrôle `HelpProvider` assure la liaison entre un fichier d'aide et l'application. Le fichier d'aide doit être généré par l'outil `Html Help Workshop`, disponible en téléchargement sur le site Microsoft. Pour nos exemples, nous utiliserons un fichier d'aide existant sur le système : le fichier C:\WINDOWS\ Help\charmap.chm correspondant à l'utilitaire Table des caractères. Ce fichier doit être associé au contrôle par la propriété `HelpNamespace`. La présence d'un contrôle `HelpProvider` sur une fenêtre, ajoute automatiquement trois propriétés à chaque contrôle présent sur la fenêtre :

`HelpKeyword`

Indique le mot clé associé au contrôle dans le fichier d'aide.

`HelpNavigator`

Indique l'action exécutée lors de l'affichage de l'aide.

`HelpString`

Contient la chaîne de caractères affichée lors de l'utilisation du bouton  d'une boîte de dialogue. Pour que ce bouton soit disponible sur la boîte de dialogue, il faut modifier la propriété`HelpButton` de la fenêtre sur **True** et masquer les boutons d'agrandissement et de réduction de la fenêtre, en modifiant les propriétés `MaximizeBox` et `MinimizeBox` sur **False**.

L'exemple suivant associe au bouton de commande `CmdOk`, la rubrique d'aide **Vue d'ensemble de la table de caractères** du fichier charmap.chm et configure le système d'aide, pour que cette rubrique soit affichée automatiquement lorsque la touche [F1] est utilisée.

HelpKeyword on HelpProvider1	**Vue d'ensemble de la table de caractères**
HelpNavigator on HelpProvider1	AssociateIndex
HelpString on HelpProvider1	**chaine affichée lors de l'utilisation du bouton ?**

h. Le contrôle ProgressBar

Ce contrôle est utilisé pour informer l'utilisateur sur la progression d'une action lancée dans l'application. Il affiche cette information sous la forme d'une zone rectangulaire, qui sera plus ou moins remplie en fonction de l'état d'avancement de l'action exécutée. L'aspect de la `ProgressBar` est contrôlé par sa propriété **Style**. Trois valeurs sont disponibles :

`Continuous`

La progression est affichée par une barre bleue pleine.

`Blocks`

La progression est affichée par une série de petits rectangles.

Marquee

Cette présentation est identique à la précédente avec, en plus, un défilement à l'intérieur de la ProgressBar.

La position de la barre de progression est contrôlée par la propriété Value. Cette propriété peut évoluée entre les deux extrêmes indiqués par les propriétés Minimum et Maximum.

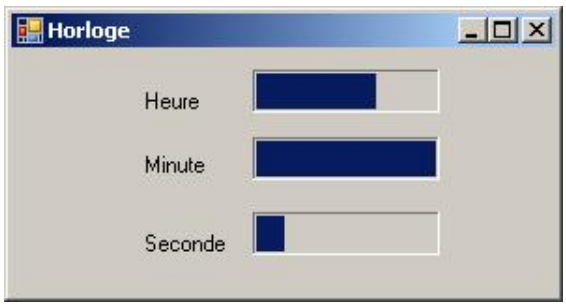
Trois techniques sont disponibles pour faire évoluer la barre de progression :

- Modifier directement la propriété Value du contrôle. À noter que dans ce cas, si la valeur de cette propriété dépasse les extrêmes, une exception est déclenchée.
- Utiliser la méthode PerformStep qui incrémente à chaque appel la propriété Value de la valeur contenue dans la propriété Step. Le contrôle vérifie, dans ce cas, la valeur contenue dans la propriété Value et s'assure qu'elle ne dépassera pas les extrêmes.
- Utiliser la méthode Increment en indiquant comme paramètre, la valeur utilisée comme incrément pour la propriété Value. La valeur de la propriété Value est également vérifiée lors de l'exécution de cette méthode.

Si la ProgressBar a le style Marquee, la propriété Value n'a aucun effet sur la taille de la barre de progression et les méthodes PerformStep et Increment ne doivent pas être utilisées, sinon une exception est déclenchée.

L'exemple suivant présente une horloge originale où l'heure est affichée par trois ProgressBar :

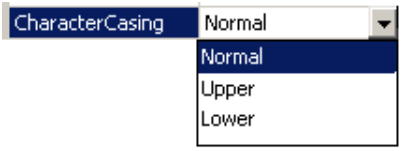
```
Public Class Horloge
    Public Sub New()
        ' This call is required by the Windows Form Designer.
        InitializeComponent()
        ' Add any initialization after the InitializeComponent() call.
        pgbHeure.Minimum = 0
        pgbHeure.Maximum = 23
        pgbHeure.Style = ProgressBarStyle.Continuous
        pgbMinute.Minimum = 0
        pgbMinute.Maximum = 59
        pgbMinute.Style = ProgressBarStyle.Continuous
        pgbSeconde.Minimum = 0
        pgbSeconde.Maximum = 59
        pgbSeconde.Style = ProgressBarStyle.Continuous
        Timer1.Interval = 500
        Timer1.Enabled = True
    End Sub

    Private Sub Timer1_Tick(ByVal sender As Object, ByVal e
As System.EventArgs) Handles Timer1.Tick
        pgbHeure.Value = Now.Hour
        pgbMinute.Value = Now.Minute
        pgbSeconde.Value = Now.Second
    End Sub
End Class
```



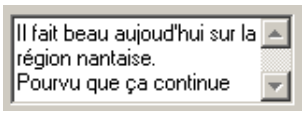
3. Les contrôles d'édition de texte

a. Le contrôle TextBox

Le contrôle `TextBox` est utilisé pour permettre à l'utilisateur de saisir des informations. Le contrôle peut être configuré pour saisir du texte sur une ou plusieurs lignes. La taille maximale du texte varie de 2000 à 32000 caractères, suivant la configuration du contrôle (simple ligne ou multiligne). Le contrôle est également capable de gérer la sélection de texte et les opérations avec le presse-papiers. De nombreuses propriétés et méthodes sont disponibles pour travailler avec ce contrôle. Le texte affiché dans le contrôle peut être modifié ou récupéré par la propriété `Text`. Le format d'affichage du texte est modifiable par différentes propriétés. La propriété `Autosize` permet de demander au contrôle `TextBox` de se redimensionner, en fonction de la taille de la police de caractères. Cette propriété est presque toujours positionnée sur **True**. La propriété `CharacterCasing` autorise le contrôle à modifier tous les caractères saisis soit en minuscules soit en majuscules.



La propriété `Lines` permet de récupérer le texte saisi, ligne par ligne. Cette propriété est un tableau de chaînes de caractères qui contient autant de cases qu'il y a de lignes. Elle n'a d'intérêt que si le contrôle est configuré pour accepter la saisie sur plusieurs lignes avec la propriété `Multiline` positionnée sur **True**. Dans ce cas, il faut également prévoir la possibilité de faire défiler le texte, en ajoutant des barres de défilement avec la propriété `ScrollBars`. Les différentes possibilités permettront d'avoir une barre de défilement horizontale, verticale ou les deux. Attention cependant car la barre de défilement verticale ne sera visible que si la propriété `WordWrap` est positionnée sur **False** sinon le contrôle gère lui-même le retour à la ligne lorsque la longueur de la ligne dépasse la largeur du contrôle. Par contre, dans ce cas les retours chariot ajoutés automatiquement ne sont pas insérés dans le texte.



Dans cet exemple, la propriété `Lines` contiendra deux éléments car le premier retour chariot est simplement ajouté par le contrôle pour l'affichage.

Lines(0)-> Il fait beau aujourd'hui sur la région nantaise

Lines(1)-> Pourvu que ça continue

La longueur maximale du texte du contrôle est fixée par la propriété`MaxLength`. À noter que dans le cas d'un contrôle multiligne, les caractères retour chariot et saut de ligne sont également comptés. Cette propriété est fréquemment utilisée lorsque l'on utilise le contrôle `TextBox` pour la saisie d'un mot de passe. Dans ce cas, la propriété `PasswordChar` indique le caractère utilisé à l'affichage, pour masquer la saisie de l'utilisateur. En général, on utilise le caractère * ou #. Cette propriété n'influence bien sûr que l'affichage, et les caractères saisis par l'utilisateur sont toujours récupérés dans la propriété `Text`.

La gestion de la sélection du texte se fait automatiquement par le contrôle. La propriété `SelectedTextpermet` de récupérer la chaîne de caractères actuellement sélectionnée dans le contrôle. Les propriétés `SelectionStart` et `SelectionLength` indiquent respectivement le caractère de début de la sélection (le premier caractère à l'indice 0) et le nombre de caractères de la sélection. Ces propriétés sont également utilisées pour insérer du texte dans le contrôle : la propriété `SelectionStart` indique, dans ce cas, le point d'insertion et la propriété `SelectedText`, le texte à insérer. Pour l'ajout de texte à la suite de celui déjà présent dans le contrôle, il est plus pratique d'utiliser la méthode AppendText en passant comme paramètre la chaîne de caractèrse à ajouter.

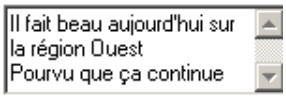
Le remplacement d'une portion de texte dans le contrôle `TextBox` se fait en deux étapes. Il faut d'abord sélectionner le texte que l'on veut remplacer à l'aide des propriétés `SelectionStart` et `SelectionLength`, puis indiquer le texte de remplacement avec la propriété `SelectedText`. Le texte remplacé et le texte de remplacement ne doivent pas forcément avoir la même taille.

```
TextBox1.SelectionStart = 39
TextBox1.SelectionLength = 8
TextBox1.SelectedText = "Ouest"
```

La sélection de texte peut également s'effectuer avec la méthode `Select`, en indiquant le caractère de début de la sélection et le nombre de caractères de la sélection.

```
TextBox1.Select(39,8)
```

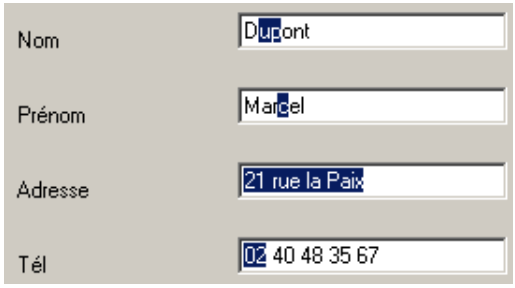
```
        TextBox1.SelectedText = "Ouest"
```



La sélection de la totalité du texte peut être effectuée avec la méthode `SelectAll`. Par exemple, on peut forcer la sélection de tout le texte lorsque le contrôle reçoit le focus.

```
Private Sub TextBox1_GotFocus(ByVal sender As Object, ByVal e As System.
EventArgs) Handles TextBox1.GotFocus
        TextBox1.SelectAll()
End Sub
```

De manière classique, dès qu'un contrôle perd le focus, la sélection de texte qu'il y avait à l'intérieur du contrôle n'est plus visible. La propriété `HideSelection` positionnée sur **False** permet de conserver la sélection visible, même si le contrôle n'a plus le focus.



Pour la gestion du presse-papiers, le contrôle `TextBox` dispose d'un menu contextuel permettant d'effectuer les opérations courantes. Vous avez cependant la possibilité d'appeler les méthodes `copy`, `cut` et `paste` pour gérer les opérations de copier coller d'une autre manière, par exemple un menu de l'application. Les opérations couper et coller ne seront cependant pas possibles si le contrôle `TextBox` est configuré en lecture seule avec la propriété `ReadOnly` positionnée sur **True**, la modification du texte par l'utilisateur est bien dans ce cas impossible.

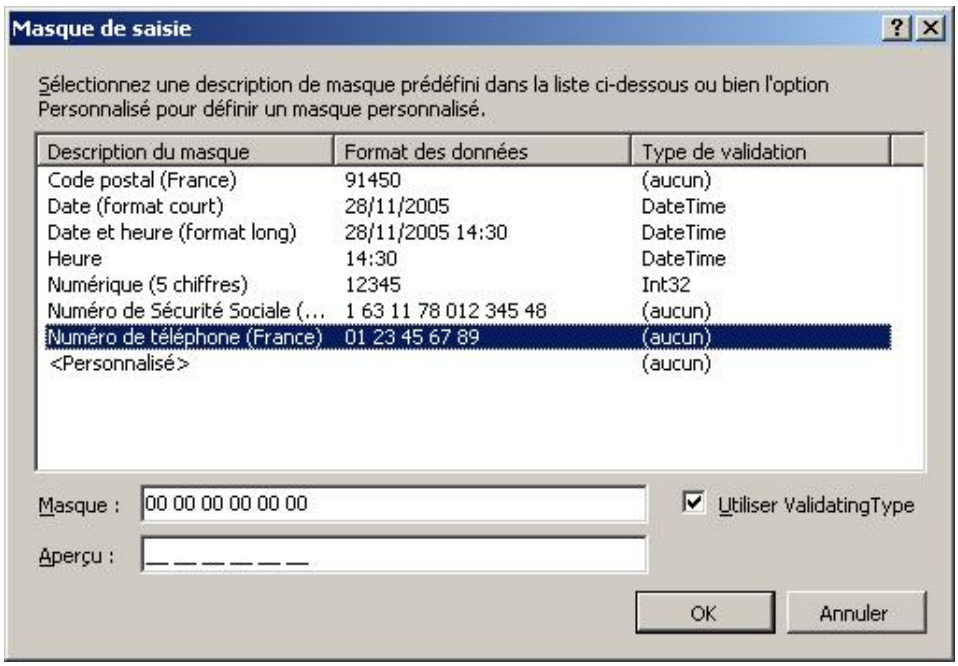
Tout le monde ayant droit à l'erreur, le contrôle `TextBox` nous propose la méthode `Undo` permettant d'annuler la dernière modification de texte effectuée sur le contrôle. Cette méthode est déjà utilisable par l'option **annuler** du menu contextuel du contrôle `TextBox` ou par le raccourci-clavier [Ctrl] **Z**. Elle peut bien sûr être appelée par un autre menu de votre application. Il n'y a qu'un seul niveau de "`Undo`", vous ne pourrez revenir au texte que vous avez saisi, il y a deux heures !

Est également disponible sur ce contrôle, l'événement `TextChanged` qui se produit lorsque la propriété `Text` du contrôle est modifiée (par le code de l'application ou par l'utilisateur). Attention à ne pas modifier la propriété `Text` de votre contrôle dans le code de cet événement sinon vous risquez de générer un débordement de pile à l'exécution de l'application, comme dans l'exemple ci-dessous.

```
Private Sub TextBox1_TextChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles TextBox1.TextChanged
        TextBox1.Text = Rnd()
End Sub
```

b. Le contrôle MaskedTextBox

Ce contrôle est une amélioration du contrôle `TextBox` car il permet de vérifier automatiquement que les informations saisies correspondent à ce qui est attendu par l'application. La propriété `Mask` détermine le format des informations pouvant être saisies dans le contrôle. L'éditeur accessible par la fenêtre de propriétés, permet de choisir un masque existant ou de configurer votre propre masque.



Pour la propriété `Mask`, certains caractères ont une signification particulière :

0 représente un chiffre obligatoire (0 à 9)

9 représente un chiffre ou un espace optionnel

L représente une lettre obligatoire (de a à z ou A à Z).

? représente une lettre optionnelle.

C représente un caractère quelconque

. représente le séparateur décimal

, représente le séparateur des milliers

: représente le séparateur horaire

/ représente le séparateur de date

$ représente le symbole monétaire

< les caractères suivants seront transformés en minuscules.

> les caractères suivants seront transformés en majuscules

| annule l'effet des deux caractères > et <

\ caractère d'échappement faisant perdre sa signification spéciale au caractère suivant

- Tous les autres caractères sont affichés tels quels dans le contrôle.

Le masque suivant peut, par exemple, être utilisé pour la saisie d'une adresse IP :

```
000\.000\.000\.000
```

c. Le contrôle RichTextBox

Le contrôle `RichTextBox` permet l'affichage, la saisie et la manipulation de texte avec mise en forme. Il possède les mêmes fonctionnalités que le contrôle `TextBox` mais il est capable de gérer des polices de caractère différentes, des

couleurs différentes, des images, etc. Il propose en fait toutes les fonctions de base d'une application de traitement de texte. Nous allons donc détailler ces principales fonctions.

Chargement et enregistrement de fichier

Les méthodes `LoadFile` et `SaveFile`permettent le chargement et l'enregistrement depuis ou vers un fichier. Le seul paramètre obligatoire pour ces deux fonctions représente le chemin d'accès complet vers le fichier à charger ou à sauvegarder. Le format de fichier utilisé par défaut par ces deux fonctions est le format rtf (*Rich Text Format*). Si d'autres formats de fichier doivent être utilisés nous devons alors le spécifier par un deuxième paramètre qui est une constante de l'énumération `RichTextBoxStreamType`. Dans le cas d'une lecture de fichier il est important que les informations contenues dans le fichier soient en accord avec la constante utilisée.

Par exemple, la lecture d'un fichier texte normal avec la ligne de code suivante déclenchera une exception.

```
rtb.LoadFile(dlgOuvrir.FileName, RichTextBoxStreamType.RichText)
```

Par contre, il n'y a pas de problème pour l'enregistrement car c'est le contrôle `RichTextBox` qui gère lui-même le format des informations inscrites dans le fichier. Le seul risque est de perdre les informations de mise en page contenues dans le contrôle si un format d'enregistrement inadapté est utilisé. Par exemple la ligne suivante fera perdre toutes les informations de formatage lors de l'enregistrement du fichier.

```
rtb.SaveFile(dlgEnregistrer.FileName, RichTextBoxStreamType.PlainText)
```

Ajout de texte

Le contrôle `RichTextBox` gère son contenu grâce à deux propriétés :

- la propriété `Text` qui, comme pour les autres contrôles, définit les informations affichées dans le contrôle. C'est le texte brut qui est stocké dans cette propriété.
- La propriété `Rtf` contient elle aussi le texte affiché par le contrôle mais elle contient en plus les informations concernant la présentation du texte.

Pour ajouter du texte à un contrôle nous pouvons définir la propriété `Text` ou la propriété `Rtf`. Dans ce second cas, il faut également inclure les caractères de formatage utilisés par le contrôle pour la mise en forme du texte. Les codes Rtf étant loin d'être simples cette solution est rarement utilisée. Heureusement une solution plus souple existe.

La méthode `AppendText` permet d'ajouter une chaîne de caractères au contenu du contrôle `RichTextBox`. Le formatage de la chaîne à ajouter doit être défini au préalable par l'intermédiaire des propriétés de la sélection.

La propriété `SelectionFont` détermine la police utilisée pour l'affichage du texte ajouté. Pour ajouter du texte en gras et souligné il faut donc modifier les caractéristiques de la police. Plus exactement il faut créer la police désirée car il n'est pas possible de modifier les caractéristiques d'une police existante puisque ses propriétés sont en lecture seule.

La syntaxe suivante est correcte pour la modification des caractéristiques de la police existante.

```
rtb.SelectionFont = New Font(rtb.SelectionFont, FontStyle.Bold +
FontStyle.Underline)
```

Les couleurs du texte et du fond sont déterminées par les propriétés `SelectionColor` et `SelectionBackColor`.

```
rtb.SelectionBackColor = Color.Beige
rtb.SelectionColor = Color.Red
```

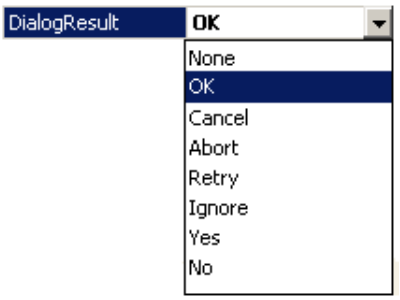
L'alignement du texte est représenté par la propriété `SelectionAlignment` à laquelle vous devez affecter une des valeurs de l'énumération `HorizontalAlignment`.

```
rtb.SelectionAlignment = HorizontalAlignment.Center
```

4. Les contrôles de déclenchement d'actions

a. Le contrôle Button

Le contrôle `Button` est principalement utilisé, dans une application, pour lancer l'exécution d'une action. Cette action peut être l'exécution d'une portion de code ou la fermeture d'une boîte de dialogue. Comme pour les contrôles vus jusqu'à présent, le libellé du bouton est modifiable par la propriété `Text` du contrôle. Cette propriété `Text` peut contenir un & pour créer un raccourci-clavier. Lorsque le bouton est utilisé pour la fermeture d'une boîte de dialogue, vous devez indiquer par l'intermédiaire de la propriété `DialogResult`, la valeur qui sera renvoyée lorsque la boîte de dialogue sera fermée par ce bouton. Les valeurs utilisables pour cette propriété, correspondent aux boutons standards rencontrés sous Windows.

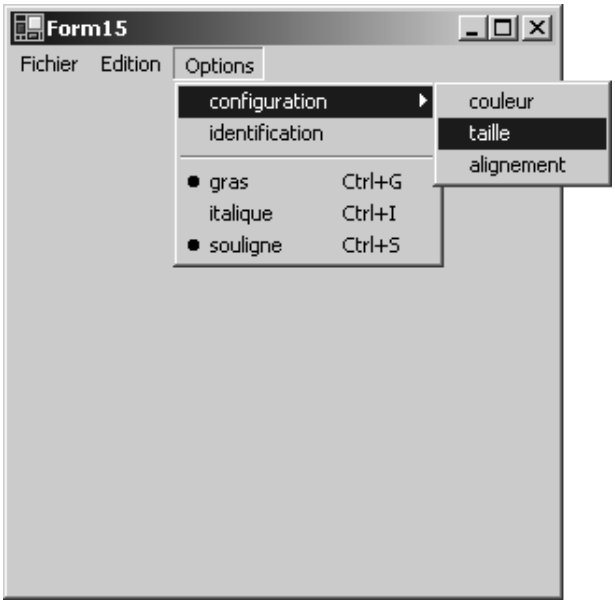


Lorsque l'on crée une boîte de dialogue, il y a généralement au moins un bouton de validation et un bouton d'annulation. On peut indiquer que, pour cette boîte de dialogue, l'utilisation de la touche [Entrée] sera équivalente à un clic sur le bouton de validation, et l'utilisation de la touche [Echap] sera équivalente à l'utilisation du bouton d'annulation. Il convient, pour cela, d'indiquer dans la propriété `AcceptButton` de la fenêtre, le nom du contrôle correspondant au bouton de validation et dans la propriété `CancelButton`, le nom du contrôle utilisé pour l'annulation.

b. Le contrôle MenuStrip

La majorité des applications disposent d'une barre de menus. Par l'intermédiaire de ces menus, l'utilisateur pourra activer les différentes fonctionnalités de l'application. Les menus sont constitués d'éléments de menus permettant le lancement d'actions dans l'application ou l'apparition d'un sous-menu. Dans ce cas, un petit triangle à côté du libellé de menu prévient l'utilisateur de la présence d'un sous-menu. Les menus peuvent également servir à activer ou non une option, dans ce cas, un petit repère est affiché devant le libellé du menu pour marquer l'activation de l'option. Pour les inconditionnels du clavier, il est également possible d'associer à un menu une combinaison de touches équivalente à un clic sur le menu.

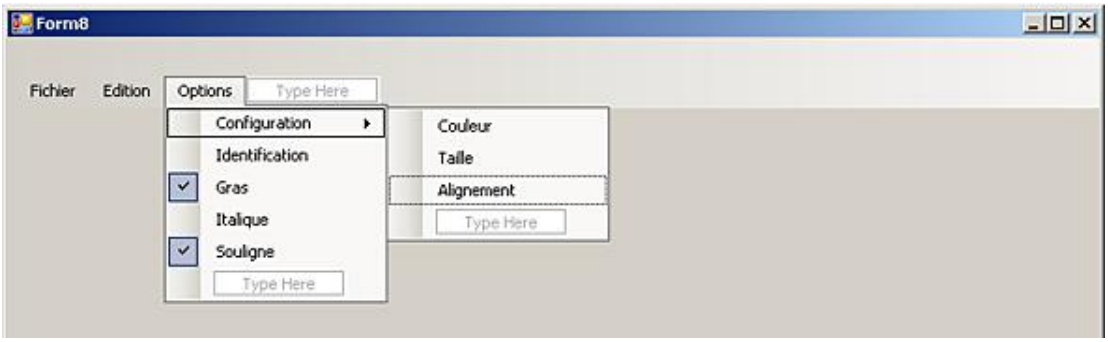
La figure suivante présente toutes ces notions.



Voyons maintenant comment mettre tout cela en œuvre. La première chose à faire est d'insérer un contrôle `MenuStrip` sur la feuille. Ce contrôle se place à deux endroits dans le concepteur :

- dans la zone réservée aux contrôles invisibles ;
- directement sur la fenêtre, à la place d'une barre de menu standard.

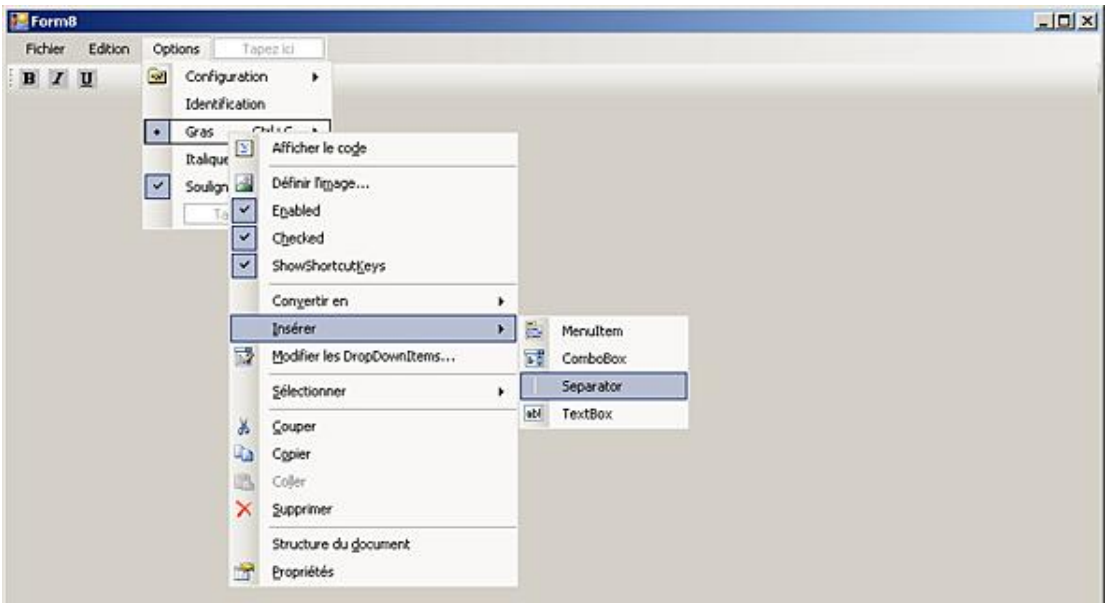
C'est à partir de cet endroit que vous allez pouvoir composer le menu. Le concepteur de menu vous propose les différentes zones dans lesquelles vous pouvez saisir les informations constituant le menu.



Vous pouvez ainsi composer complètement la barre de menu et visualiser son apparence au fur et à mesure de sa conception.

À chaque fois que vous insérez un élément, le concepteur crée une nouvelle instance de la classe`ToolStripMenuItem` et l'insère dans la collection **Items**, pour les titres de menus et dans la collection**DropDownItems**, pour les éléments de menu et de sous-menu.

Le menu contextuel, affiché par un clic droit, permet l'ajout d'une barre de séparation.



Il reste maintenant à configurer les propriétés de chacun des éléments de menu.

Regardons les propriétés les plus intéressantes :

`Text`

Contient le libellé du menu.

`Visible`

Permet de masquer un élément de menu.

`Enabled`

Permet d'interdire l'utilisation de l'élément. Il apparaît alors en grisé dans le menu.

`Checked`

Indique si l'élément de menu est coché ou non.

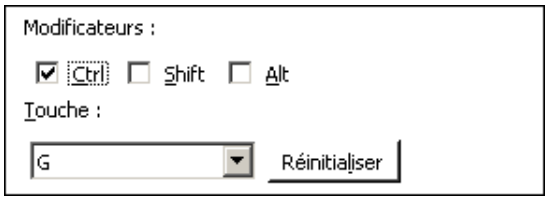
`CheckOnClick`

Permet de changer automatiquement l'état du menu à chaque clic. La valeur de la propriété `Checked` est, dans ce cas, inversée à chaque clic sur le menu.

`ShortcutKeys`

Définit la combinaison de touches équivalente à un clic sur le menu.

Cette propriété est facilement modifiable par la fenêtre de propriétés :



La propriété `ShowShortCutKeys` autorise l'affichage du raccourci à côté du libellé du menu.

Ces propriétés sont aussi modifiables par le code. Si, par exemple, une barre d'outils est également disponible, les actions réalisées avec celle-ci doivent bien sûr agir sur l'état du menu correspondant et la réciproque doit aussi être vraie.

```
Private Sub btnGras_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles btnGras.Click
        MnuGras.Checked = btnGras.Checked
    End Sub
    Private Sub btnItalique_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnItalique.Click
        MnuItalique.Checked = btnItalique.Checked
    End Sub

    Private Sub btnSouligne_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnSouligne.Click
        MnuSouligne.Checked = btnSouligne.Checked
    End Sub

    Private Sub MnuGras_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles MnuGras.Click
        btnGras.Checked = MnuGras.Checked
    End Sub

    Private Sub MnuItalique_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MnuItalique.Click
        btnItalique.Checked = MnuItalique.Checked
    End Sub

    Private Sub MnuSouligne_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MnuSouligne.Click
        btnSouligne.Checked = MnuSouligne.Checked
    End Sub
```

Dans certains cas, il est nécessaire d'ajouter dynamiquement des éléments pendant le fonctionnement de l'application pour ajouter par exemple la liste des derniers fichiers utilisés dans l'application.

La solution consiste à ajouter, à la collection **DropDownItems**, un nouvel élément en utilisant la méthode `Add`.

```
MnuFichier.DropDownItems.Add("essai.txt")
```

Cette opération ajoute bien un élément au menu, mais il ne nous sera pas d'une grande utilité ! Il nous est en effet impossible de pouvoir récupérer les événements `Click` sur ce menu.

Pour résoudre ce problème, nous devons utiliser une syntaxe différente de la méthode `Add` nous permettant d'associer un gestionnaire d'événement au nouvel élément ajouté.

Il faut d'abord créer la procédure qui sera utilisée pour la gestion de cet événement :

```
Private Sub FichiersRecent_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)
```

```
End Sub
```

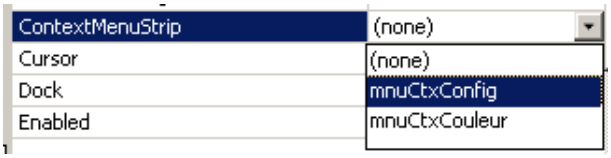
Puis, au moment d'ajouter l'élément de menu, nous devons indiquer que notre procédure sera responsable de la gestion des événements `Click` sur le menu.

```
MnuFichier.DropDownItems.Add("essai.txt", Nothing, New System.EventHandler
(AddressOf FichiersRecent_Click))
```

c. Le menu ContextMenuStrip

Le menu contextuel permet d'associer aux différents contrôles de l'interface de l'application, un menu qui sera affiché lors d'un clic avec le bouton droit de la souris sur le contrôle. Les menus doivent d'abord être conçus en ajoutant sur la feuille des contrôles `ContextMenuStrip`.

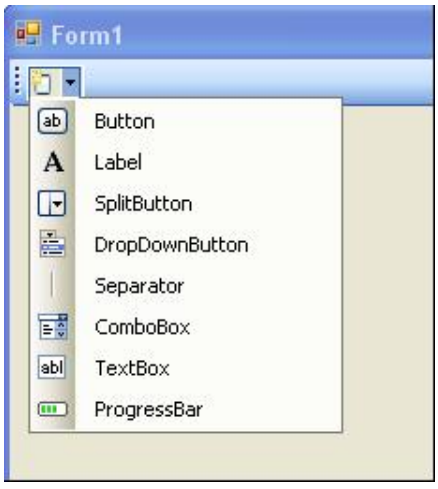
Ce contrôle apparaît, sur la barre de menu de l'application, juste pour vous permettre de saisir les différents éléments du menu. À noter qu'un menu contextuel ne comporte pas de titre mais permet simplement la saisie d'éléments de menu et de sous-menus. Il faut ensuite associer, à chaque contrôle, le menu contextuel correspondant par l'intermédiaire de la propriété `ContextMenuStrip`.



L'affichage du menu se fera automatiquement lorsque l'utilisateur fera un clic droit sur le contrôle correspondant.

d. Le contrôle ToolStrip

Le contrôle `ToolStrip` et toutes les classes associées sont utilisés pour la création de barres d'outils. Ce contrôle sert en fait de conteneur pour les éléments constituant une barre d'outils. Ces éléments sont ajoutés au contrôle `ToolStrip` par l'intermédiaire du menu qui est affiché en cliquant sur la petite flèche présente à gauche sur le contrôle.



Cette solution permet de créer une barre d'outils de A à Z en y ajoutant tous les éléments un à un. Dans beaucoup d'applications, les barres d'outils comportent un ensemble de boutons standard. Pour nous faciliter la tâche, le contrôle `ToolStrip` contient un assistant pour générer automatiquement une barre d'outils avec toutes les fonctionnalités de base d'une application. Vous devez pour cela cliquer sur la petite flèche présente sur la droite du contrôle et choisir l'option **Ajouter des éléments standard**. La barre d'outils prend alors l'apparence suivante.

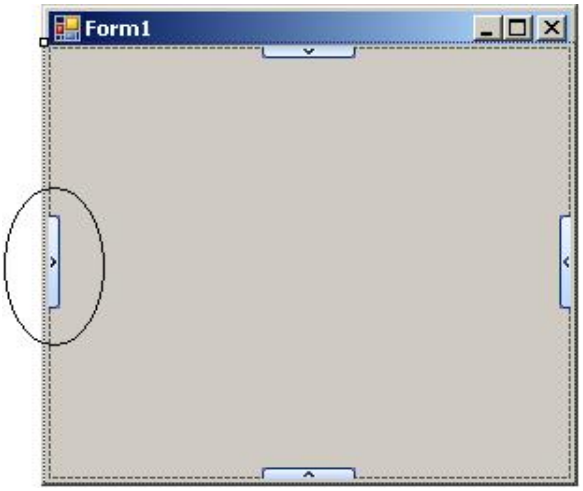
Les boutons classiques sont ajoutés automatiquement à la barre d'outils. La 'magie' de l'assistant s'arrête ici car il n'y a, bien sûr, pas de code généré automatiquement pour réagir aux clics sur ces différents boutons. Vous devez donc

l'écrire vous-même comme pour tous les autres contrôles de l'interface de l'application.

e. Le contrôle ToolStripContainer

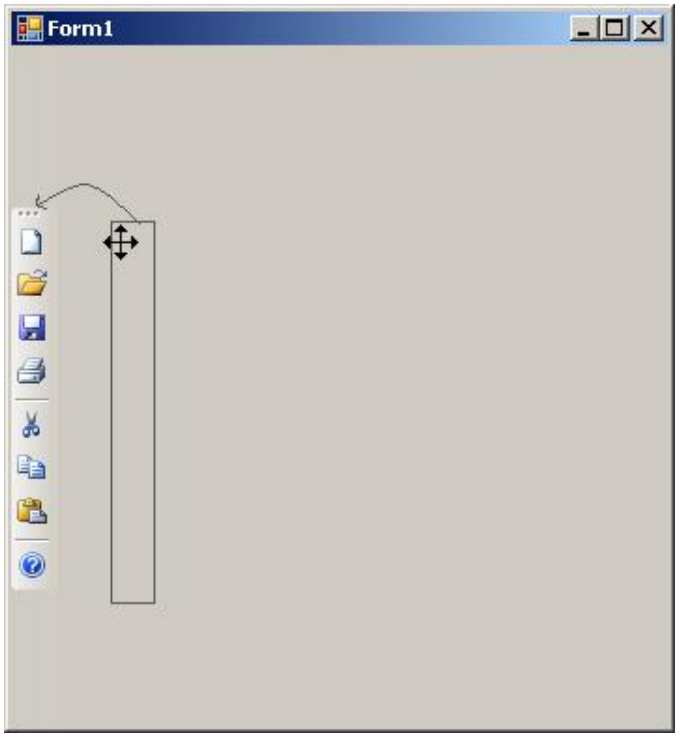
Le but de ce contrôle est de faciliter la conception d'une fenêtre lorsque celle-ci doit contenir plusieurs barres d'outils. Ce contrôle est généralement placé sur une fenêtre avec la propriété `Dock` positionnée sur `Fill` pour occuper toute la surface disponible de la fenêtre.

Il dispose sur chacune de ses bordures d'un conteneur pour accueillir les différentes barres d'outils. Par défaut ces conteneurs ne sont pas visibles et il faut donc les activer avant de pouvoir y ajouter des éléments. Pour cela, vous devez cliquer sur les petites flèches présentes sur les onglets en périphérie du contrôle.



Une nouvelle zone devient active sur laquelle il est possible de placer un ou plusieurs `ToolStrip`. Attention ces différentes zones ne peuvent contenir que des contrôles `ToolStrip`. La zone centrale peut par contre contenir n'importe quel contrôle.

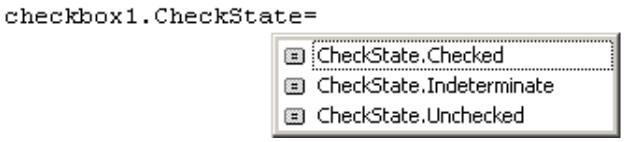
L'avantage de l'utilisation du contrôle `ToolStripContainer` réside dans la possibilité de modifier la disposition des barres d'outils pendant l'exécution de l'application. Pour déplacer une barre d'outils, l'utilisateur doit la saisir en cliquant sur les trois points présents sur la barre d'outils et réaliser un glisser déplacer vers une des bordures de la fenêtre. La barre d'outils vient alors s'ancrer sur la bordure correspondante.



5. Contrôles de sélection

a. Le contrôle CheckBox

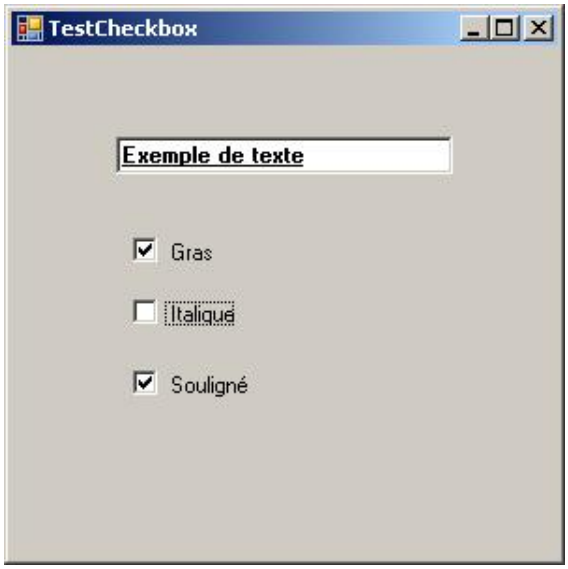
Le contrôle CheckBox est utilisé pour proposer à l'utilisateur plusieurs options, parmi lesquelles il pourra en choisir une ou plusieurs. Le contrôle CheckBox peut en général prendre deux états : coché lorsque l'option est sélectionnée ou non coché lorsque l'option n'est pas sélectionnée. Une troisième possibilité correspond à un état indéterminé de la case à cocher ; dans ce cas, elle apparaît en grisé. L'état de la case à cocher peut être vérifié ou modifié par la propriété Checked. Cette propriété fournit un boolean qui reflète l'état de la case. Si la case à cocher est configurée pour fonctionner avec trois états possibles, en positionnant la propriété ThreeState sur **True**, la propriété CheckState indique l'état de la case à cocher par l'intermédiaire d'une des valeurs définies dans l'énumération.



Comme pour les autres contrôles, la propriété TextAlign permet de modifier la position du texte sur le contrôle. La propriété CheckAlign, suivant le même principe, permet de modifier la position de la case à cocher par rapport au texte.

Par défaut, le contrôle CheckBox gère lui-même l'affichage de la coche et la modification des propriétés Checked et CheckState en fonction des actions de l'utilisateur. Vous pouvez reprendre la responsabilité de la gestion de l'état de la case à cocher, en modifiant la propriété AutoCheck sur **False**. Dans ce cas, vous devez gérer l'événement Click et modifier par le code, dans la gestion de cet événement, la propriété Checked ou la propriété CheckState. L'exemple de code, ci-après, permet de modifier les caractéristiques de la police de caractère d'un contrôle TextBox.

Dans cet exemple, nous travaillons avec l'événement CheckedChanged du contrôle CheckBox qui nous indique chaque changement de l'état de la case à cocher.



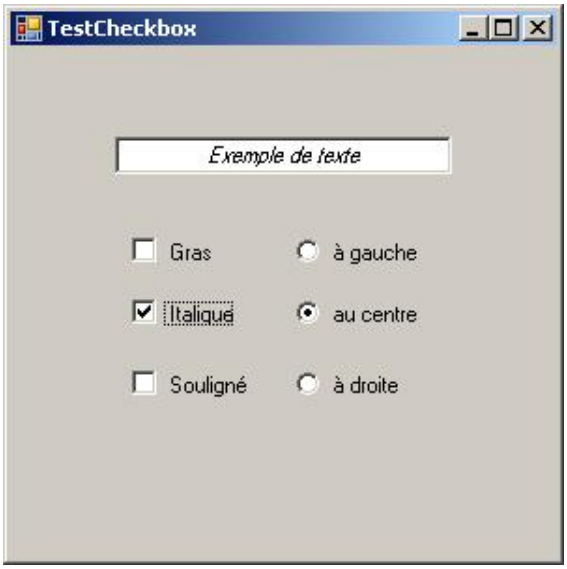

```
Private Sub chkGras_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles chkGras.CheckedChanged
        Dim style As Integer
        style = TxtExemple.Font.Style
        If chkGras.Checked Then
            style = style Or FontStyle.Bold
        Else
            style = style And Not FontStyle.Bold
        End If
        TxtExemple.Font = New Font(TxtExemple.Font, style)
End Sub

Private Sub chkItalique_CheckedChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles chkItalique.CheckedChanged
        Dim style As Integer
        style = TxtExemple.Font.Style
        If chkItalique.Checked Then
```

```
            style = style Or FontStyle.Italic
        Else
            style = style And Not FontStyle.Italic
        End If
        TxtExemple.Font = New Font(TxtExemple.Font, style)
 End Sub

Private Sub chkSouligne_CheckedChanged(ByVal sender As System.Object,  ByVal e
As System.EventArgs) Handles chkSouligne.CheckedChanged
        Dim style As Integer
        style = TxtExemple.Font.Style
        If chkSouligne.Checked Then
            style = style Or FontStyle.Underline
        Else
            style = style And Not FontStyle.Underline
        End If
        TxtExemple.Font = New Font(TxtExemple.Font, style)
End Sub
```

b. Le contrôle RadioButton

Le contrôle `RadioButton` permet également de proposer à l'utilisateur différentes options parmi lesquelles il ne pourra en sélectionner qu'une seule. Comme son nom l'indique, ce contrôle fonctionne comme les boutons permettant de sélectionner une station sur un poste de radio (Vous ne pouvez pas écouter trois stations de radio en même temps !). Les propriétés sont strictement identiques à celles disponibles dans le contrôle `CheckBox` hormis la propriété `CheckState` car ce contrôle ne peut avoir que deux états et un boolean est suffisant pour représenter l'état du contrôle.

Complétons notre exemple précédent en ajoutant des options pour l'alignement du texte :




```
Private Sub optGauche_CheckedChanged(ByVal sender As System.Object,
By Val e As System.EventArgs) Handles optGauche.CheckedChanged
        If optGauche.Checked Then
            TxtExemple.TextAlign = HorizontalAlignment.Left
        End If
End Sub

Private Sub optDroite_CheckedChanged(ByVal sender As System.Object,
By Val e As System.EventArgs) Handles optDroite.CheckedChanged
        If optDroite.Checked Then
            TxtExemple.TextAlign = HorizontalAlignment.Right
        End If
End Sub

Private Sub optCentre_CheckedChanged(ByVal sender As System.Object,
By Val e As System.EventArgs) Handles optCentre.CheckedChanged
        If optCentre.Checked Then
```

```
            TxtExemple.TextAlign = HorizontalAlignment.Center
End If
```

Il est conseillé, lorsque l'interface de votre application contient des contrôles `RadioButton`, qu'il y en ait toujours un de sélectionné à l'affichage du formulaire (correspondant à l'option par défaut).

Complétons notre application en ajoutant le choix de la couleur du texte :


```
Private Sub optNoir_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles optNoir.CheckedChanged
        TxtExemple.ForeColor = System.Drawing.Color.Black
End Sub
Private Sub optVert_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles optVert.CheckedChanged
        TxtExemple.ForeColor = System.Drawing.Color.GreenYellow
End Sub

Private Sub optBleu_CheckedChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles optBleu.CheckedChanged
        TxtExemple.ForeColor = System.Drawing.Color.Blue
End Sub
```

Nous avons quelques soucis pour le fonctionnement de notre application, car il n'est pas possible de choisir une option d'alignement et une option de couleur simultanément. Les contrôles `RadioButton` étant conçus de telle sorte qu'ils ne puissent pas y en avoir deux sélectionnés simultanément dans un même conteneur. La solution à notre problème passe donc par l'utilisation d'un conteneur pour isoler les contrôles les uns des autres. C'est ce que nous verrons avec les contrôles `GroupBox` et `Panel`.

c. Le contrôle ListBox

Le contrôle `ListBox` propose à l'utilisateur une liste de choix dans laquelle il pourra en sélectionner un ou plusieurs. Le contrôle gère automatiquement l'affichage des éléments avec, au besoin, l'ajout d'une barre de défilement sur le contrôle. Les éléments affichés dans la liste peuvent être saisis au moment de la conception, dans la fenêtre de propriétés, ou modifiés par le code. En règle générale, les éléments de la liste sont des chaînes de caractères mais peuvent être n'importe quel type d'objet. Par défaut, le contrôle `ListBox` propose les éléments sous forme de liste avec un défilement vertical.

La propriété `MultiColumn` positionnée sur **True** permet d'avoir un défilement horizontal de la liste.

Dans ce cas, la propriété `ColumnWidth` indique la largeur de chacune des colonnes. La propriété `IntegralHeight` évite

d'avoir un élément de la liste partiellement visible.

Si cette propriété est positionnée sur **True**, la liste se redimensionne pour n'afficher que des éléments complets.

Les éléments de la liste sont gérés sous forme de collection, par la propriété `Items`. Nous pouvons donc ajouter des éléments à la liste en utilisant la méthode `Add` de la collection et en passant, comme paramètre, l'objet à insérer dans la liste. Dans la majorité des cas, l'objet inséré est une chaîne de caractères mais n'importe quel autre type d'objet peut être utilisé. Il faut cependant que l'objet ajouté dispose d'une méthode `ToString` pour permettre l'affichage par le contrôle `ListBox`.

```
Public Class couleur
    Public nom As String
    Public valeur As System.Drawing.Color

    Public Sub New(ByVal n As String, ByVal v As System.Drawing.Color)
        nom = n
        valeur = v
    End Sub

    Public Overrides Function toString() As String
        Return nom
    End Function

End Class


...
...
LstCouleurs.Items.Add(New couleur("rouge", System.Drawing.Color.Red))
LstCouleurs.Items.Add(New couleur("vert", System.Drawing.Color.Green))
```

Les éléments sont ajoutés à la fin de la liste sauf si la propriété `Sorted` est positionnée sur **True** car, dans ce cas, les éléments de la liste sont triés par ordre croissant. La méthode `Insert` ajoute un élément dans la liste, à un emplacement précis (le premier élément de la liste se trouve à l'emplacement zéro). Pour insérer un élément en deuxième position, nous utiliserons le code ci-après :

```
LstCouleurs.Items.Insert(1, New couleur("Jaune",System.Drawing.Color.Yellow))
```

L'effacement d'un élément de la liste s'effectue avec la méthode `RemoveAt` en passant comme paramètre l'index de l'élément à supprimer.

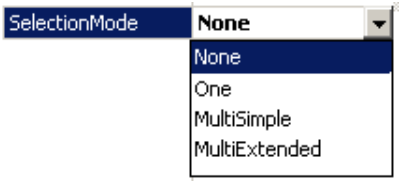
```
LstCouleurs.Items.RemoveAt(0)
```

On peut aussi effectuer un nettoyage complet de la liste avec la méthode `Clear`.

```
LstCouleurs.Items.Clear()
```

Deux méthodes sont utiles, lors de l'ajout d'éléments dans la liste, si l'on veut éviter les doublons. La méthode `FindString` recherche le premier élément de la liste qui commence par la chaîne de caractères spécifiée. Cette fonction renvoie l'index de l'élément correspondant au critère ou la valeur -1, si rien n'est trouvé. La même chose peut être effectuée avec la méthode `FindString Exact` mais, dans ce cas, il faut une concordance exacte des chaînes de caractères.

Par défaut, les `ListBox` n'autorisent que la sélection d'un élément. La propriété `SelectionMode` permet de choisir quatre modes de sélection :



- Le premier (**None**) risque d'être frustrant pour l'utilisateur car la `ListBox` sera pleinement fonctionnelle mais le clic souris ne produira aucun effet.

- L'option **One** est la valeur par défaut où chaque clic de souris sélectionne l'élément et désélectionne l'élément précédemment sélectionné dans la liste.

- Avec l'option **MultiSimple**, chaque clic sur un élément inverse son état. Il peut y avoir plusieurs éléments sélectionnés en même temps.

- La dernière option **MultiExtended** permet également la sélection de plusieurs éléments mais offre la possibilité de sélectionner plusieurs éléments successifs rapidement. En sélectionnant le premier, puis en utilisant la touche [Shift] et en cliquant sur le dernier, tous les éléments présents entre les deux sont également sélectionnés.

Il faut bien sûr récupérer, dans le code, la ou les sélections de l'utilisateur. Dans le cas d'une sélection simple, la propriété`SelectedIndex` indique l'indice, dans la collection **Items**, de l'élément sélectionné alors que la propriété `SelectedItem` fournit une référence vers l'objet sélectionné dans la liste.

Pour les sélections multiples, le principe de fonctionnement est le même mais les informations sont renvoyées sous forme de collection. La collection**SelectedIndices** nous propose la liste des indices des éléments sélectionnés, alors que la collection **SelectedItems** nous donne la liste des objets sélectionnés. Dans les deux cas, il faut parcourir la liste pour extraire les éléments.

```
        Dim o As Object
        For Each o In ListBox1.SelectedItems
            System.Console.WriteLine(o)
        Next
```

Le fait de stocker des objets dans la collection **Items** nous permet d'avoir accès, par l'intermédiaire de la propriété `SelectedItem`, à toutes les propriétés de l'objet. Nous pouvons ainsi, par exemple, dans le cas où nous avons utilisé des objets couleurs, utiliser la propriété `valeur`.

Dans l'exemple suivant, nous modifions la couleur de fond de la feuille en fonction de la sélection dans la `ListBox`. Pour cela, nous gérons l'événement`SelectedIndexChanged` qui se produit à chaque nouvelle sélection de l'utilisateur.

```
Private Sub LstCouleur_SelectedIndexChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles ListBox2.SelectedIndexChanged
        Me.BackColor = ListBox2.SelectedItem.valeur
End Sub
```

d. Le contrôle NumericUpDown

Le contrôle `NumericUpDown` est l'association d'une zone de texte et de deux boutons. Ces deux boutons servent à incrémenter ou décrémenter la valeur affichée dans la zone de texte. Ce contrôle est très simple à utiliser. La valeur qu'il est chargé de gérer est stockée dans la propriété `value`. Elle peut évoluer entre deux extrêmes représentés par les propriétés `Minimum` et `Maximum`. Le pas d'incrémentation est défini par la propriété `Increment`. Toutes ces valeurs peuvent être des valeurs décimales. Le format d'affichage peut être modifier par la propriété `DecimalsPlaces` qui permet de spécifier le nombre de décimales affichées. Cette propriété intervient uniquement au niveau de l'affichage et ne change pas la valeur de la propriété `Value`. Voici comme exemple un petit morceau de code qui simule le fonctionnement d'un thermostat de chauffage. Ce n'est bien sûr pas la température de votre bureau qui sera modifiée par l'application, mais la couleur de fond de la fenêtre du bleu pour le plus "froid" au rouge pour le plus "chaud".

```
Private Sub NumericUpDown1_ValueChanged(ByVal sender As Object, ByVal e
As System.EventArgs) Handles Thermostat.ValueChanged
    Me.BackColor = Color.FromArgb(25 * (Thermostat.Value
- 15), 0, 25 * (25 - Thermostat.Value))
End Sub

Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Load
        Me.Thermostat.DecimalPlaces = 1
        Me.Thermostat.Increment = 0.1
        Me.Thermostat.Maximum = 25
        Me.Thermostat.Minimum = 15
        Me.Thermostat.Value = 15
End Sub
```

e. Le contrôle TrackBar

Ce contrôle propose un aspect plus visuel du contrôle NumericUpDown tout en fournissant les mêmes fonctionnalités. Il se présente sous la forme d'un potentiomètre sur lequel l'on peut déplacer un curseur pour ajuster une valeur numérique. Contrairement au contrôle NumericUpDown, il n'y a pas d'affichage de la valeur et celle-ci ne peut être qu'une valeur entière. Les deux valeurs extrêmes sont définies par les propriétés Minimum et Maximum.

Deux pas d'incrémentation sont également disponibles avec les propriétés SmallChange et LargeChange. La propriété SmallChange est utilisée pour incrémenter ou décrémenter la valeur lors de l'utilisation des flèches de déplacement du clavier. La propriété LargeChange est utilisée suite à la frappe des touches page suivante ou page précédente. Si le curseur du potentiomètre est déplacé à l'aide de la souris, la propriété Value correspond à l'emplacement du curseur. Aucun des deux pas d'incrémentation n'est utilisé dans ce cas. Pour faciliter l'ajustement d'une valeur, des repères sont placés sur le potentiomètre. Leur espacement est déterminé par la propriété TickFrequency.

f. Le contrôle DomainUpDown

Ce contrôle a pratiquement le même fonctionnement que le contrôle Listbox. Il se présente comme une zone de texte à laquelle sont associés deux boutons permettant le déplacement dans une liste d'éléments. La principale propriété de ce contrôle est la propriété Items qui contient les éléments proposés dans la liste. Les seules différences avec le contrôle Listbox sont liées aux sélections multiples qui sont impossibles pour ce contrôle et la propriété Wrap permettant le parcours de la liste de manière circulaire.

g. Le contrôle CheckedListBox

Ce contrôle est une amélioration du contrôle ListBox et de ce fait fonctionne pratiquement de la même manière. La seule limitation par rapport au contrôle ListBox est qu'il n'accepte pas les sélections multiples. L'évolution par rapport au contrôle ListBox réside dans les cases à cocher présentes devant les éléments de la liste. Ces cases peuvent être cochées indépendamment de la sélection des éléments. La récupération des cases cochées est possible par l'intermédiaire de la propriété CheckedItems. Cette collection contient tous les éléments cochés de la liste.

```
For Each elt In CheckedListBox1.CheckedItems
            Console.WriteLine(elt.ToString)
Next
```

Une autre solution pour obtenir les éléments cochés de la liste consiste à parcourir la liste et à tester avec la méthode GetItemChecked si la case correspondante est cochée.

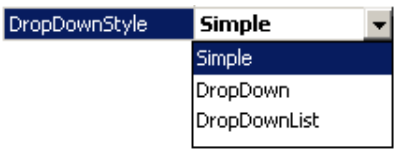
```
For i = 0 To CheckedListBox1.Items.Count - 1
     If CheckedListBox1.GetItemChecked(i) Then
           Console.WriteLine(CheckedListBox1.Items(i))
     End If
Next
```

La propriété CheckOnClick détermine si l'activation ou la désactivation de la case à cocher se fait sur le premier clic d'un élément dans la liste (true) ou si un double clic est nécessaire pour changer l'état de la case à cocher (false).

h. Le contrôle ComboBox

Le contrôle ComboBox est l'association d'un contrôle ListBox et d'un contrôle TextBox. L'utilisateur pourra, suivant la configuration du contrôle, choisir un élément dans la liste ou saisir du texte.

La propriété DropDownStyle indique le mode de fonctionnement de la ComboBox.



Simple

L'utilisateur sélectionne un élément dans la liste ou saisit du texte dans la zone de texte. La liste est affichée en permanence.

DropDown

Même configuration, mais la liste n'est affichée qu'à la demande de l'utilisateur.

`DropDownList`

L'utilisateur ne peut plus saisir dans la zone de texte mais simplement choisir un élément de la liste, après avoir demandé son affichage.

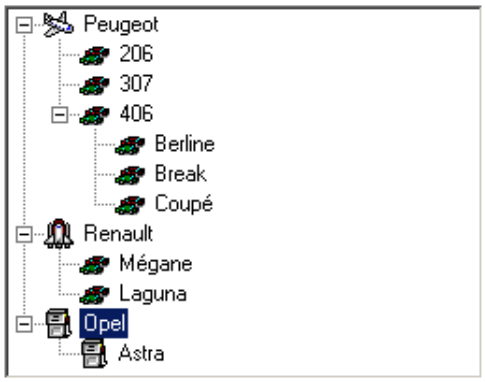
L'état de l'affichage de la liste est contrôlé par la propriété `DroppedDown`. On peut, par exemple, écrire du code pour que la liste soit déroulée dès que le contrôle reçoit le focus.

```
Private Sub ComboBox1_GotFocus(ByVal sender As Object, ByVal e As System.
EventArgs) Handles ComboBox1.GotFocus
     ComboBox1.DroppedDown = True
End Sub
```

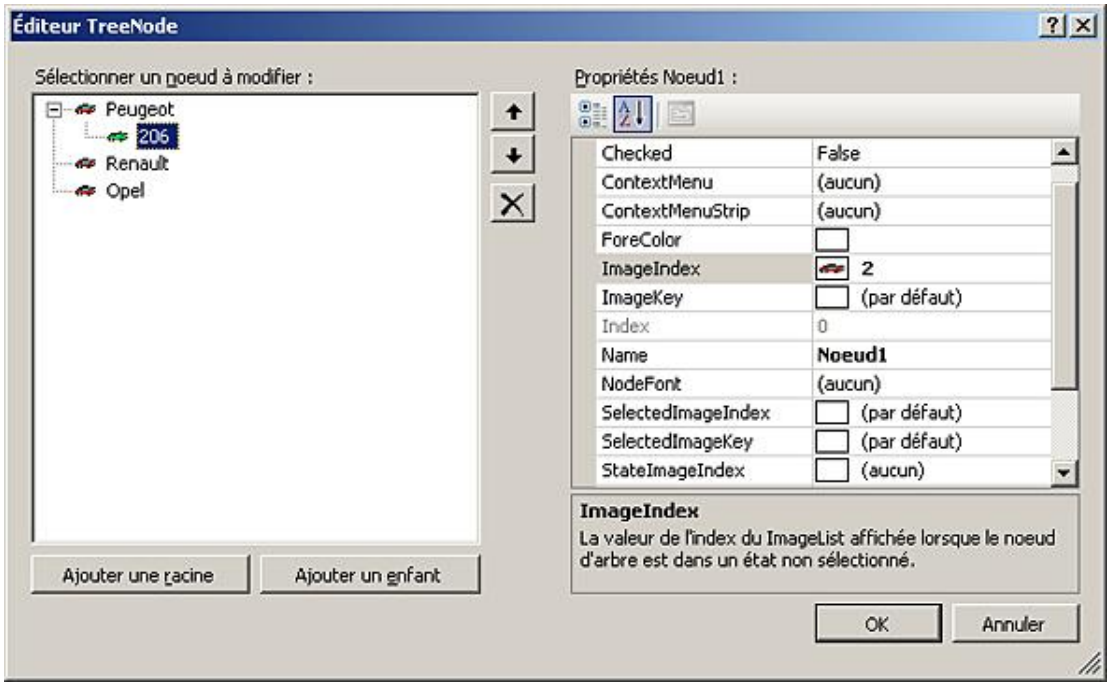
Il n'est pas possible de réaliser des sélections multiples avec le contrôle `ComboBox`. Le texte peut être récupéré dans la propriété `Text`. Cette propriété est mise à jour même si l'utilisateur a sélectionné un élément dans la liste. Dans ce cas, la propriété `SelectedIndex` est égale à l'index de l'élément sélectionné. Si l'utilisateur a saisi du texte directement, cette propriété est égale à -1.

i. Le contrôle TreeView

Le contrôle `TreeView` permet la présentation d'informations, sous forme d'une arborescence identique à la présentation des fichiers et des dossiers dans l'explorateur de Windows. Chaque élément est représenté sous forme d'un nœud pouvant éventuellement contenir des nœud enfants. Chaque élément peut être affiché sous forme développée ou réduite, par l'intermédiaire du signe (plus (+) ou moins (-)) affiché en face de chaque nœud.



La propriété `Nodes` contient la liste de tous les nœuds de premier niveau de l'arborescence. Chacun d'entre eux dispose également d'une propriété`Nodes`, qui stocke à son tour la liste de tous ses nœuds enfants et ainsi de suite jusqu'au dernier niveau de l'arborescence. Ces propriétés `Nodes` sont modifiables au moment de la conception, par un éditeur spécifique accessible par la fenêtre de propriétés.



Le bouton **Ajouter une racine** ajoute un élément racine à la propriété `Nodes` du contrôle `TreeView`. Pour chaque élément, il convient également d'indiquer son libellé par la propriété `Text`, l'image affichée sur le nœud par la propriété `ImageIndex` ainsi que l'image affichée, lorsque le nœud est sélectionné dans l'arborescence, par la propriété `SelectedImageIndex`.

Ces images sont extraites du contrôle `ImagelList` que vous aurez, au préalable, associé à la propriété `Imagelist` du contrôle. Ce contrôle `ImageList` doit bien sûr être créé et rempli avant la modification des propriétés du contrôle `TreeView`.

Chaque nœud possède donc une image, pour son affichage normal, et une image, pour son affichage lorsqu'il est sélectionné. La valeur (**Default**) pour ces deux images indique que le nœud n'utilisera pas une image spécifique, mais les images par défaut qui sont indiquées dans les propriétés `ImageIndex` et `SelectedImageIndex` du contrôle `TreeView`. Vous pouvez de cette façon utiliser des images identiques pour tous les nœuds. Seuls ceux ayant un affichage particulier nécessiteront la modification des valeurs par défaut.

L'ajout de nœud dans l'arborescence par le code est un petit peu plus complexe. Il faut créer un nouveau nœud en instanciant la classe `TreeNode` et en indiquant, dans l'appel du constructeur de la classe, la légende affichée sur le nœud. L'image et l'image de sélection sont également à préciser si vous ne voulez pas utiliser les valeurs par défaut. La création d'un nœud peut donc prendre les deux formes suivantes.

```
        Dim noeud As TreeNode
        noeud = New TreeNode("Twingo")
```

ou bien

```
        Dim noeud As TreeNode
        noeud = New TreeNode("Twingo ", 2, 3)
```

en indiquant, dans ce cas, l'index de l'image pour l'affichage normal, suivi de l'index pour l'affichage lorsque le nœud sera sélectionné.

Il faut ensuite "raccrocher" ce nouveau nœud à une branche de notre arborescence, en utilisant méthode `Add` du nœud qui va devenir le parent du petit nouveau.

Ainsi, pour ajouter un enfant à la deuxième racine de l'arborescence, nous utiliserons le code suivant :

```
TreeView1.Nodes(1).Nodes.Add(noeud)
```

Vous pouvez aussi supprimer un nœud par le code, en utilisant la méthode `RemoveAt` de la collection correspondante. Par exemple, pour supprimer la Peugeot 206, nous utiliserons le code suivant :

```
TreeView1.Nodes(0). Nodes.RemoveAt(0)
```

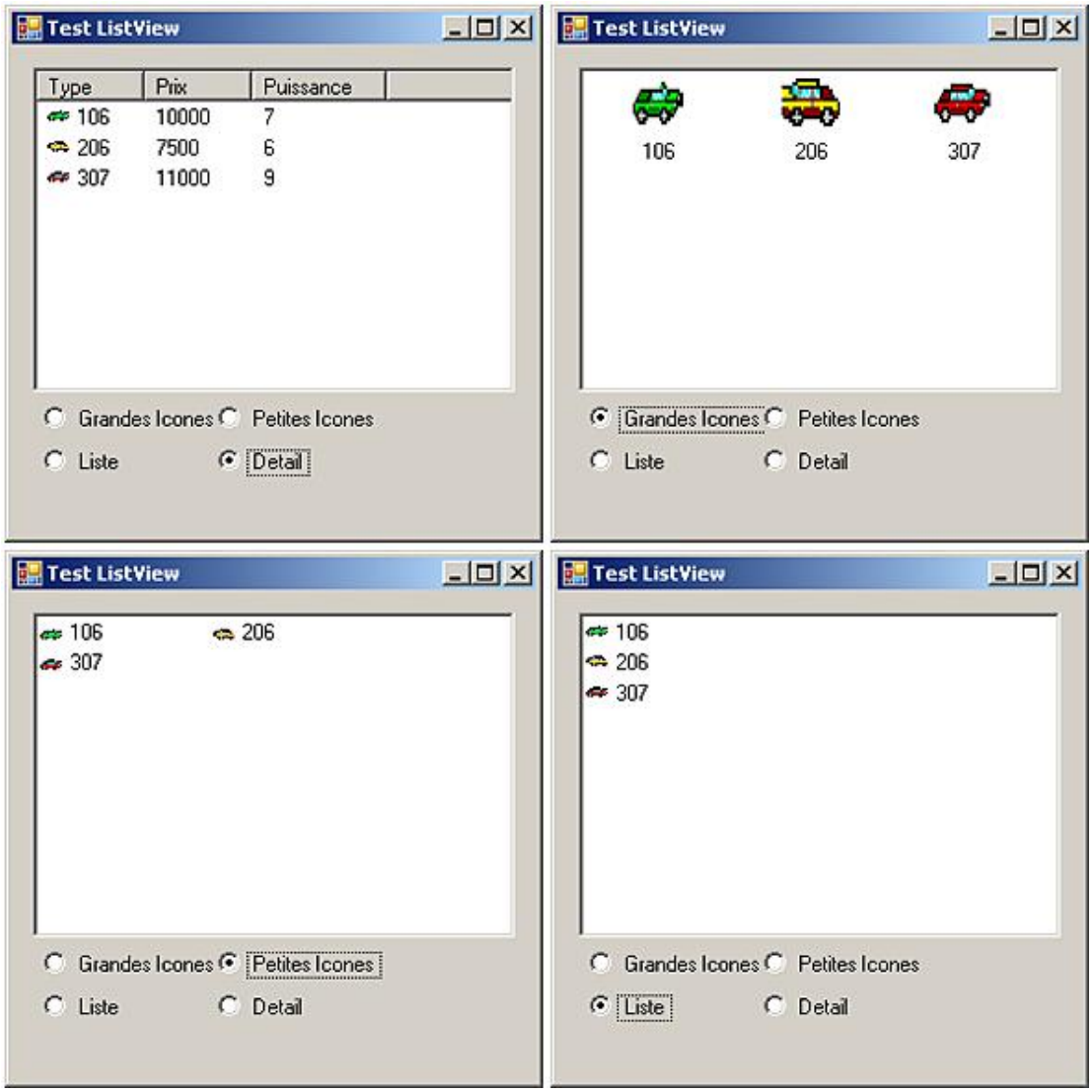
Lorsque l'on travaille avec le contrôle `TreeView`, on a fréquemment besoin de savoir sur quel élément du contrôle l'utilisateur a cliqué (pour afficher les caractéristiques du véhicule dans notre cas).

Dans l'événement `AfterSelect` du contrôle `TreeView`, on dispose, par l'intermédiaire du paramètre `e`, une référence sur le nœud sur lequel l'utilisateur a cliqué. La propriété `Text` de cet élément nous permet donc d'identifier le nœud du contrôle `TreeView` et de réagir en conséquence.

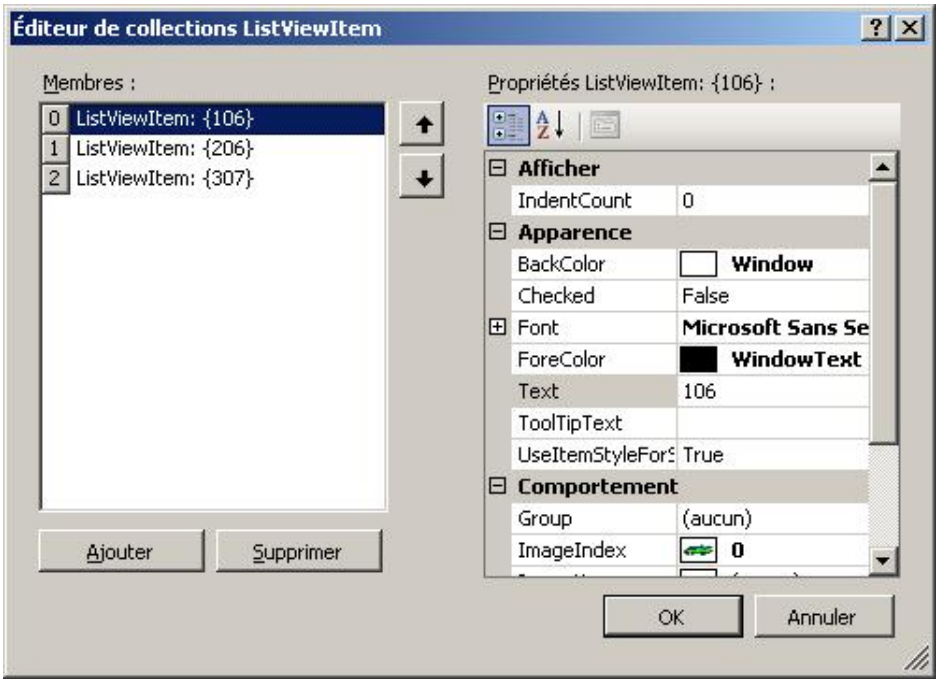
```
Private Sub TreeView1_AfterSelect(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.TreeViewEventArgs) Handles TreeView1.AfterSelect
        Select Case e.Node.Text
            Case "Peugeot"
                System.Console.WriteLine("Peugeot")
            Case "Renault"
                System.Console.WriteLine("Renault")
            Case "Opel"
                System.Console.WriteLine("Opel")
        End Select
End Sub
```

j. Le contrôle ListView

Ce contrôle fournit la possibilité de présenter des informations, à l'utilisateur, de quatre façons différentes comme le permet le volet droit de l'explorateur de Windows.

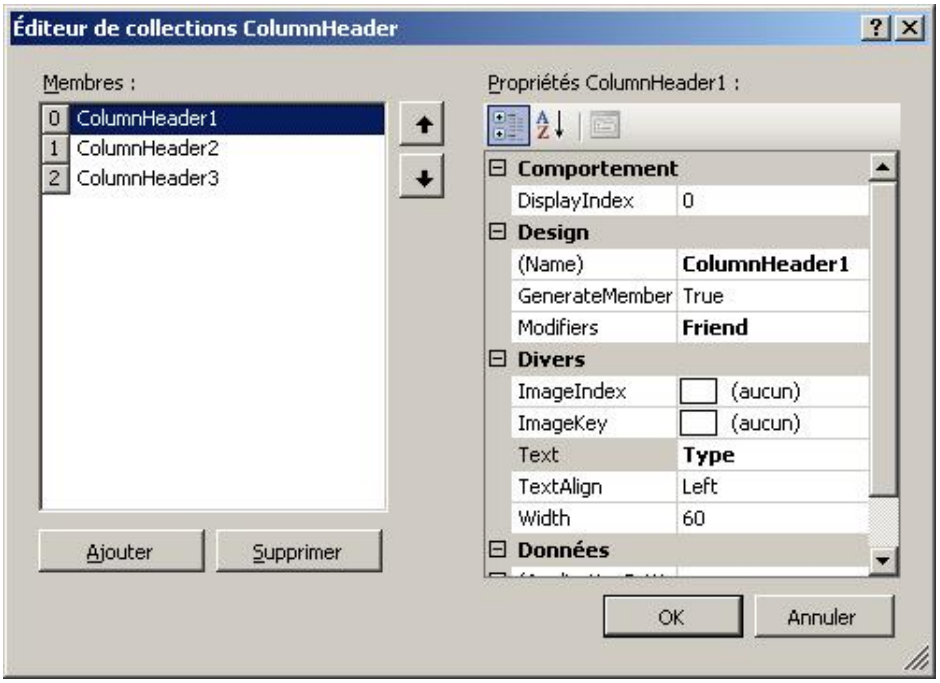


Comme pour le contrôle `TreeView`, ce contrôle dispose d'une collection **Items** stockant les informations à afficher. Cette collection peut être remplie par un éditeur spécifique accessible depuis la fenêtre de propriétés de vb.



Cet éditeur permet d'ajouter ou de supprimer des éléments dans l'affichage du contrôle et pour chacun d'entre eux, la modification de la légende par la propriété `Text` et de l'image associée par la propriété `ImageIndex`. La propriété`SubItems` est également une collection utilisée lors de l'affichage détaillé. Cette collection contient les informations affichées dans chacune des colonnes du mode détaillé.

Les colonnes sont créées par la propriété `Columns` du contrôle. Un éditeur spécifique est également disponible pour la gestion des colonnes.



La propriété `Text` correspond au titre des colonnes. Chaque colonne est associée aux éléments qui auront été saisis dans la propriété `SubItems`. Résumons tout cela avec la présentation de nos véhicules et de leurs caractéristiques.

Pour chaque véhicule à afficher, il faut insérer un élément dans la collection **Items** du contrôle `ListView`. Pour chacun de ces éléments, la propriété `SubItems` doit contenir les informations utilisées pour l'affichage en mode détail (le type du véhicule, son prix et sa puissance).

Il faut ensuite ajouter trois colonnes à la propriété `Columns` du contrôle `ListView` (type, prix, puissance).

Pour terminer la configuration, les propriétés `LargeImageList` et `SmallImageList` indiquent respectivement les contrôles `ImageList` où seront récupérées les images pour l'affichage en mode Grandes Icônes ou Petites Icônes.

Le type de présentation de l'affichage est modifiable par la propriété `View` :

```
Private Sub optGrande_CheckedChanged(ByVal sender As System.Object, ByVal
 e As System.EventArgs) Handles optGrande.CheckedChanged
        ListView1.View = View.LargeIcon
    End Sub

    Private Sub optPetites_CheckedChanged(ByVal sender As System.Object,
ByVal eAs System.EventArgs) Handles optPetites.CheckedChanged
        ListView1.View = View.SmallIcon
    End Sub

    Private Sub optListe_CheckedChanged(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles optListe.CheckedChanged
        ListView1.View = View.List
    End Sub

    Private Sub optDetail_CheckedChanged(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles optDetail.CheckedChanged
        ListView1.View = View.Details
    End Sub
```

6. Les contrôles de regroupement

a. Le contrôle GroupBox

Le contrôle `GroupBox` permet le regroupement logique et visuel de contrôles sur un formulaire. Le contrôle possède une bordure et une légende apparaissant sur la bordure haute. Ce contrôle facilite également le placement des contrôles sur la fenêtre, car après avoir placé les contrôles sur le `GroupBox`, ceux-ci se déplacent en bloc avec le contrôle `GroupBox`. Il permet également d'isoler les `RadioButtons`, les uns des autres, lorsqu'ils doivent fonctionner de manière indépendante.

b. Le contrôle Panel

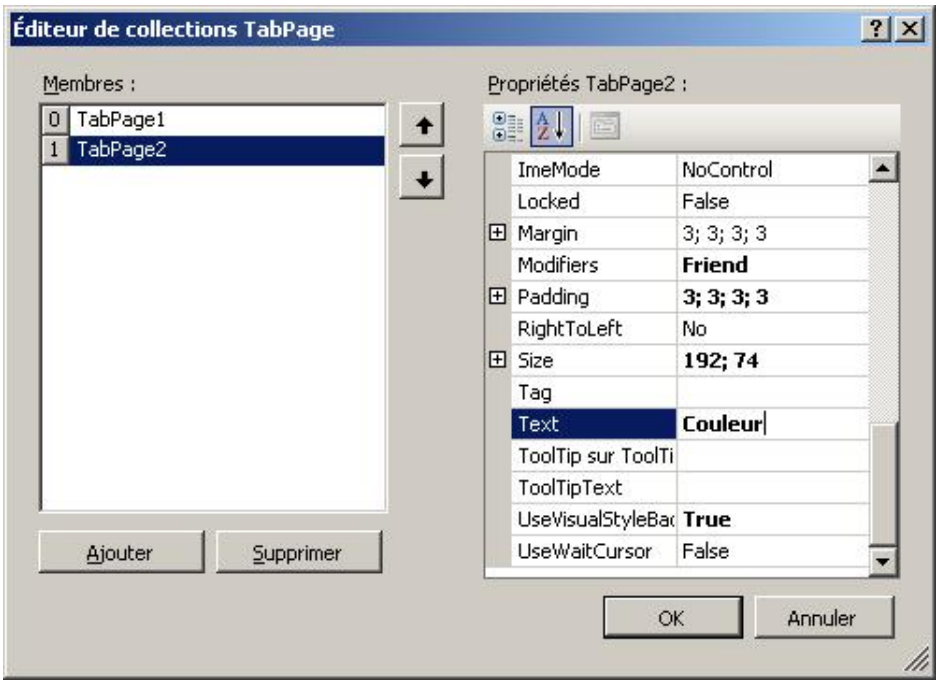
Le contrôle `Panel` dispose des mêmes fonctionnalités que le contrôle `GroupBox`. Il ne dispose cependant pas de propriété `Text` et ne permet donc pas l'affichage d'une légende. Par contre, il possède une propriété `BorderStyle` permettant de supprimer l'affichage de la bordure. Nous avons, dans ce cas, des contrôles qui sont regroupés logiquement sans indication visible de ce regroupement. La propriété `AutoScroll` du contrôle permettra d'afficher automatiquement des barres de défilement si sa surface n'est pas suffisante pour afficher l'ensemble des contrôles qui lui sont confiés.

Pour ces deux contrôles, la modification de la propriété `Enabled` désactivera également l'ensemble des contrôles qui sont placés à l'intérieur, sans avoir à modifier cette propriété pour chacun des contrôles.

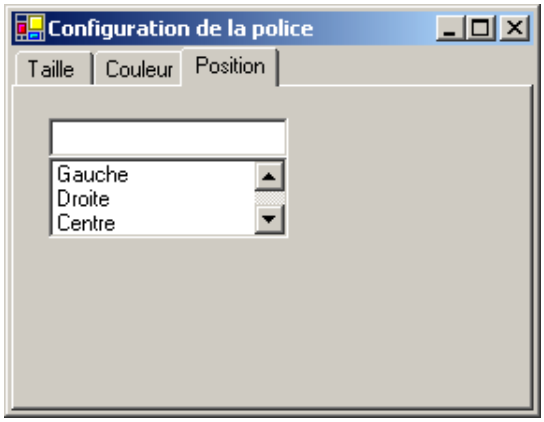
c. Le contrôle TabControl

Le contrôle `TabControl` affiche plusieurs onglets similaires aux intercalaires d'un classeur. L'utilisation principale de ce contrôle est la création de boîte de dialogue à plusieurs onglets.

La propriété principale `TabPages` contient la liste de toutes les pages associées au contrôle. Un éditeur spécifique nous permet la manipulation de ces pages.



Chaque onglet est considéré comme une page, sur laquelle on pourra placer des contrôles. Pour chacun des onglets, on retrouve d'ailleurs beaucoup de propriétés déjà disponibles sur les fenêtres.



L'insertion de contrôles s'effectue en sélectionnant au préalable l'onglet de destination, puis en dessinant les contrôles dessus, de la même manière que sur une fenêtre classique. Si votre boîte de dialogue contient de nombreux onglets, vous pouvez choisir de les afficher sur plusieurs lignes en modifiant la propriété `Multiline` du contrôle.

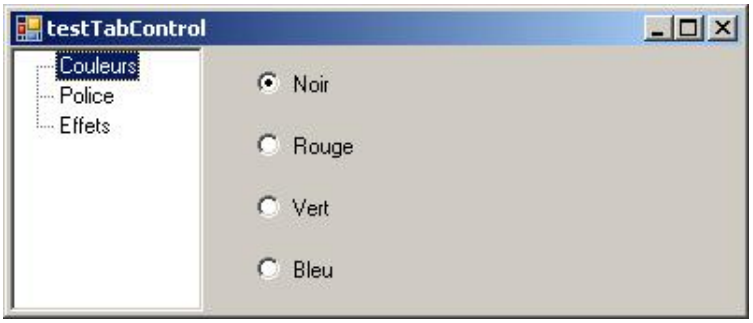
Il est également possible d'interdire l'utilisation de tous les contrôles présents sur un onglet, en modifiant la propriété `Enabled` de l'onglet. Par exemple, pour interdire l'utilisation de l'onglet **Couleur** :

```
TabControl1.TabPages(1).Enabled = False
```

Comme pour beaucoup d'autres contrôles, une propriété `ImageList` permet d'associer des images bitmap à chacun des onglets, par l'intermédiaire de la propriété `ImageIndex` de chaque onglet.

d. Le contrôle SplitContainer

Ce contrôle permet la création d'interface utilisateur comparable à l'explorateur Windows. Il sépare la fenêtre en deux parties par une barre horizontale ou verticale. Cette barre peut être déplacée par l'utilisateur pour redimensionner chacune des zones délimitées. Il est fréquemment associé avec un contrôle `TreeView` dans sa partie gauche et un ensemble de contrôles dans la partie droite permettant la modification de l'élément sélectionné dans le TreeView. Cette présentation est par exemple utilisée dans l'outil **MMC** (*Microsoft Management Console*), par lequel de nombreux paramètres de fonctionnement du système sont configurables. Chaque partie du contrôle est utilisable comme un containeur pour d'autres contrôles.

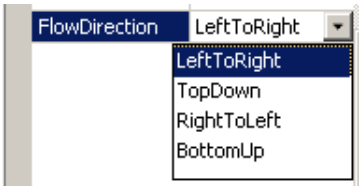


La propriété `Orientation` indique le sens dans lequel est découpé la surface du contrôle. Le dimensionnement de chacune des deux zones peut être interdit avec la propriété `IsSplitterFixed` positionnée sur **True**. Il est également possible d'interdire le redimensionnement d'une seule zone avec la propriété`FixedPanel`. Dans ce cas, si les dimensions du conteneur du contrôle (la fenêtre) sont modifiées, une seule zone est redimensionnée. La propriété `Orientation` détermine bien sûr le sens de découpage du contrôle.

e. Le contrôle FlowLayoutPanel

Le contrôle `FlowLayoutPanel` organise automatiquement le placement des contrôles que vous lui confiez. En fonction de la valeur de la propriété `FlowDirection`, les contrôles seront disposés :

- de la gauche vers la droite ;
- de la droite vers la gauche ;
- du haut vers le bas ;
- du bas vers le haut.

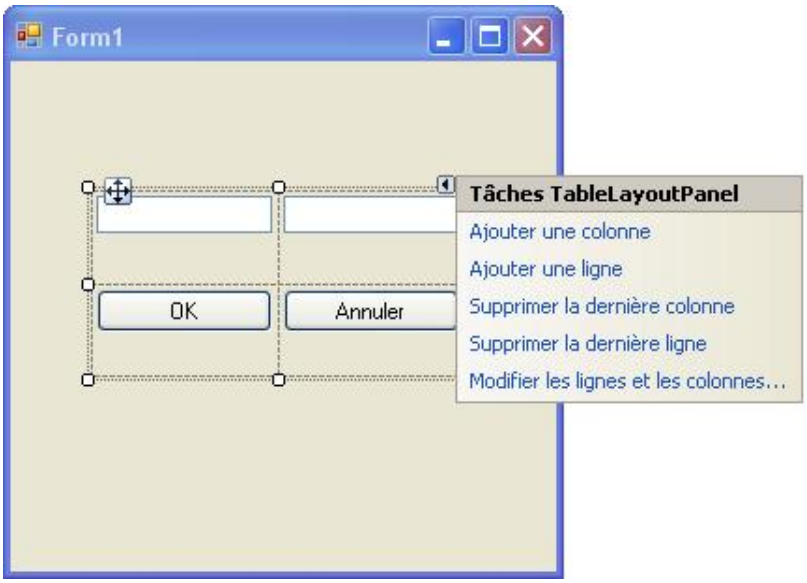


Si l'espace disponible n'est pas suffisant pour l'ajout d'un contrôle supplémentaire, la propriété`WrapContents` indique si une nouvelle colonne ou ligne est ajoutée, en fonction de l'orientation dans laquelle le contrôle `FlowLayoutPanel` travaille ou si les contrôles supplémentaires sont tronqués.

f. Le contrôle TableLayoutPanel

Il est parfois important qu'un formulaire conserve un aspect correct lorsqu'il est redimensionné. La première solution nous venant à l'esprit consiste à gérer un des événements survenant pendant le dimensionnement de la fenêtre, `Resize`, `Layout` par exemple et de modifier les positions et dimensions des contrôles en conséquence. Cette solution est très lourde à mettre en œuvre et doit être répétée sur chaque formulaire. L'autre solution, plus efficace est de demander à un gestionnaire de positionnement de gérer la taille et position des contrôles qu'on lui confie. Le contrôle `TableLayoutPanel` est spécialisé pour ce travail. Ce conteneur organise son contenu sous forme d'une grille un petit peu de la même manière qu'un tableau en langage HTML.

Il faut tout d'abord placer sur le formulaire un contrôle `TableLayoutPanel`. Par défaut le contrôle est généré avec deux lignes et deux colonnes. Pour modifier cette disposition il suffit simplement d'activer les options du contrôle en cliquant sur la petite flèche située sur le coin supérieur droit puis en utilisant l'option correspondante.



Chaque ligne et colonne peut également être modifiée après sa création en choisissant l'option **Modifier les lignes et les colonnes**.

Les contrôles peuvent ensuite être placés dans chacune des cases. Il ne peut y avoir qu'un seul contrôle par case.

Le principal intérêt de ce contrôle se situe pendant le dimensionnement du formulaire puisque les contrôles vont suivre automatiquement les modifications de taille du formulaire. Pour cela, le contrôle `TableLayoutPanel` doit être ancré sur les bordures gauche et droite du formulaire.

Il en va de même pour tous les contrôles placés dans le `TableLayoutPanel`.

Après une modification de la taille de la fenêtre, nous conservons toujours un aspect correct du formulaire.

Pour comprendre l'intérêt du contrôle `TableLayoutPanel` essayez la manipulation suivante. Ajoutez un nouveau formulaire puis faites un copier-coller des deux zones de texte et des deux boutons sur ce formulaire. Exécutez l'application et modifiez la taille du formulaire. Le résultat prouve bien l'utilité de ce contrôle.

7. Les contrôles graphiques

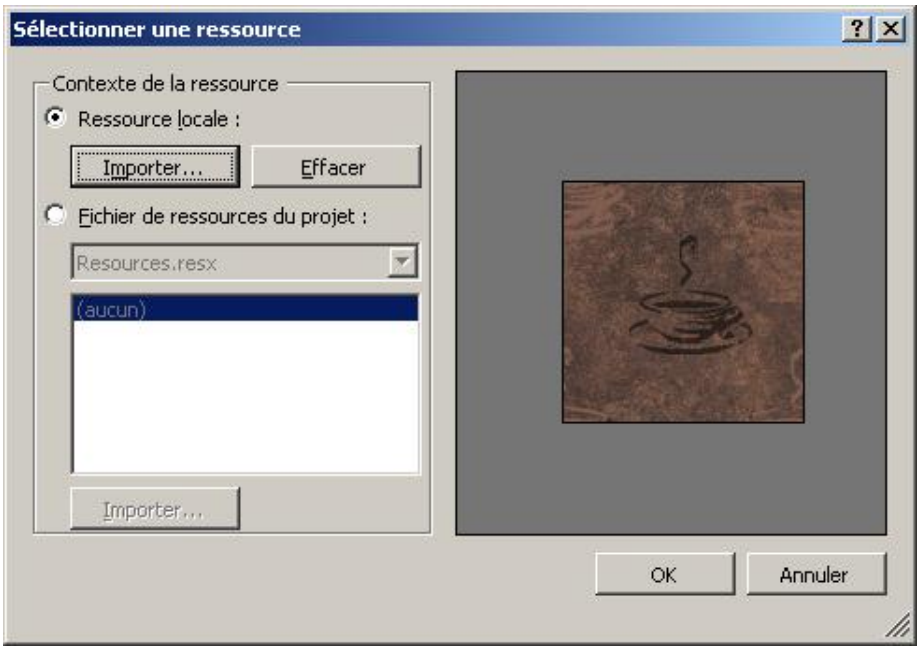
a. Le contrôle PictureBox

Le contrôle `PictureBox` est utilisé pour afficher des images dans une application. Plusieurs formats de fichiers sont pris en charge par le contrôle :

- Les fichiers Bitmap (*.bmp)

- Les fichiers Icones (*.ico)
- Les fichiers Gif (*.gif)
- Les métafichiers (*.wmf)
- Les fichiers JPEG (*.jpg)

Le contenu du contrôle est spécifié dans la propriété `Picture`. Dans la fenêtre de propriétés, vous pouvez rechercher le fichier à charger dans le contrôle grâce à la boîte de dialogue ci-après.



Le bouton **Effacer** permet de réinitialiser la propriété pour éliminer un fichier déjà présent.

Pour modifier l'image affichée, par l'intermédiaire du code, nous devons charger le contenu du fichier et l'affecter à la propriété `Picture` qui attend une instance de la classe `Image`. Pour créer cette instance, nous utilisons la méthode statique `FromFile` de la classe `Image` qui prend comme paramètre le nom du fichier et qui renvoie l'instance de la classe `Image` créée.

```
PictureBox1.Image = Image.FromFile("titi.gif")
```

Pour effacer une image par le code, vous pouvez utiliser le code suivant :

```
PictureBox1.Image = Nothing
```

Par l'intermédiaire de la propriété `SizeMode`, vous pouvez choisir comment le contrôle et l'image vont adapter leur taille respective :

normal

Les deux éléments conservent leurs dimensions. L'image est placée dans le coin supérieur gauche du contrôle.

StretchImage

L'image est agrandie pour occuper toute la surface du contrôle. Dans le cas d'images non vectorielles, le résultat est parfois décevant.

AutoSize

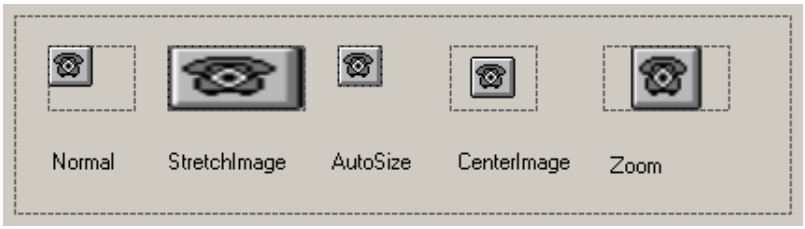
Le contrôle s'adapte à la taille de l'image.

CenterImage

Les deux éléments conservent leurs dimensions mais l'image est centrée par rapport au contrôle.

Zoom

L'image est agrandie ou rétrécie pour occuper toute la largeur ou hauteur du contrôle mais le même ratio est utilisé verticalement et horizontalement.



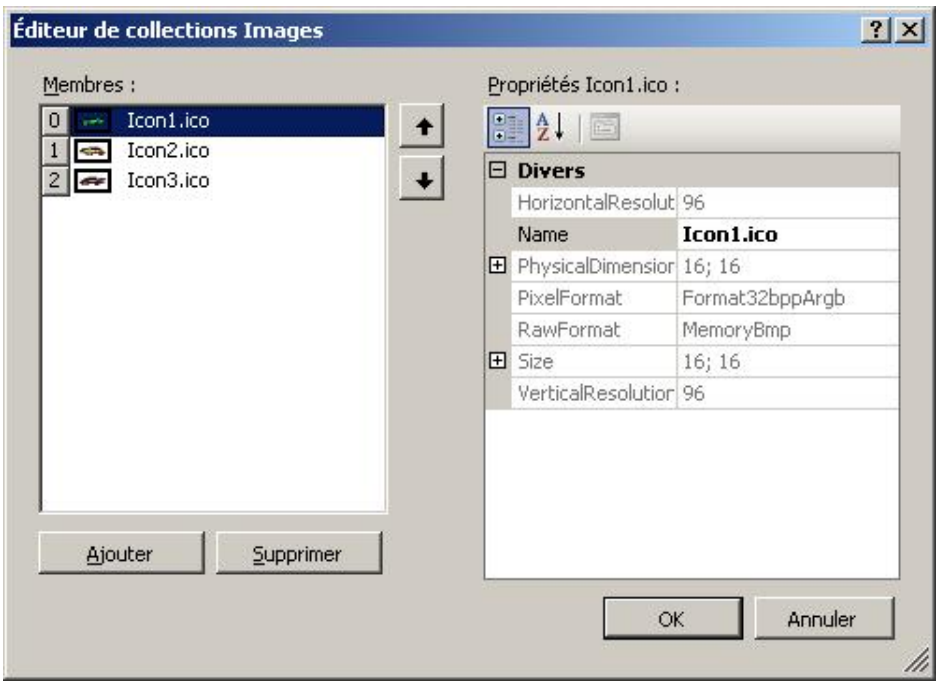
b. Le contrôle ImageList

Le contrôle `ImageList` est utilisé dans une application pour servir de "réservoir d'images". Il ne fournit aucune fonctionnalité pour l'affichage des images, mais permet simplement leur stockage dans l'application. De nombreux types de fichiers peuvent être utilisés dans ce contrôle (bmp, ico, gif, jpg...). Toutefois, les images stockées dans ce contrôle ne sont pas forcément du même type. Vous pouvez, par exemple, mélanger des fichiers bmp et des fichiers ico. Une petite restriction tout de même : les images issues de metafile (wmf) ne sont pas prises en compte. Ce contrôle n'a pas d'interface visible au moment de l'exécution donc il ne sera pas placé directement sur la feuille, mais dans une zone réservée à ce type de contrôles dans l'interface de conception graphique.

Ce contrôle sera utilisé en association avec tout type de contrôle possédant une propriété `ImageList`, `SmallImageList`, ou `LargeImageList`. Pour ces contrôles, une autre propriété, en général la propriété `ImageIndex`, indique l'image stockée dans le contrôle `ImageList` que vous voulez lui associer. De cette manière, vous pouvez rapidement modifier l'image affichée sur un bouton en modifiant la propriété `ImageIndex` du bouton, afin de sélectionner l'une des images disponibles dans le contrôle `ImageList`.

La propriété `Images` est la propriété principale du contrôle `ImageList`. C'est une collection dans laquelle les images sont stockées. Chaque image est accessible par l'intermédiaire de son index dans la collection.

Cette propriété peut être modifiée au moment de la conception de l'application, par l'intermédiaire de la fenêtre de l'éditeur de collections d'images affichée lorsque l'on modifie la propriété `Images` du contrôle.



Par l'intermédiaire de cet éditeur, on peut notamment ajouter des images, en supprimer ou modifier leur classement dans la collection. Cependant, il n'y a pas possibilité de modifier l'image une fois insérée dans la collection. Il faut, dans ce cas, la supprimer et en insérer une nouvelle avec le contenu du fichier correspondant.

L'insertion d'une image peut également s'effectuer par l'intermédiaire du code, en utilisant la méthode `Add` disponible dans la collection Images du contrôle.

```
ImageList1.Images.Add(Image.FromFile("dome.bmp"))
```

De la même façon, on peut supprimer, par le code, une image déjà présente dans la liste en utilisant la méthode `RemoveAt` et en indiquant l'index de l'élément à supprimer.

```
ImageList1.Images.RemoveAt(6)
```

La méthode `Clear` permet de vider complètement la collection `Images` supprimant ainsi toutes les images du contrôle `ImageList`.

Pour toutes les images de la liste, vous pouvez également indiquer par l'intermédiaire de la propriété `ColorDepth` le nombre de couleurs gérées par le contrôle `ImageList`. La propriété `ImageSize` indique la taille des images. Cette taille sera identique pour toutes les images placées dans le contrôle. Les images insérées seront éventuellement agrandies ou diminuées pour s'adapter à cette taille (les résultats sont parfois décevants).

8. Les contrôles de gestion du temps

Deux contrôles sont disponibles pour la saisie de dates. Le contrôle `DateTimePicker` permet la saisie d'une date alors que le contrôle `MonthCalendar` autorise la saisie d'une plage de dates.

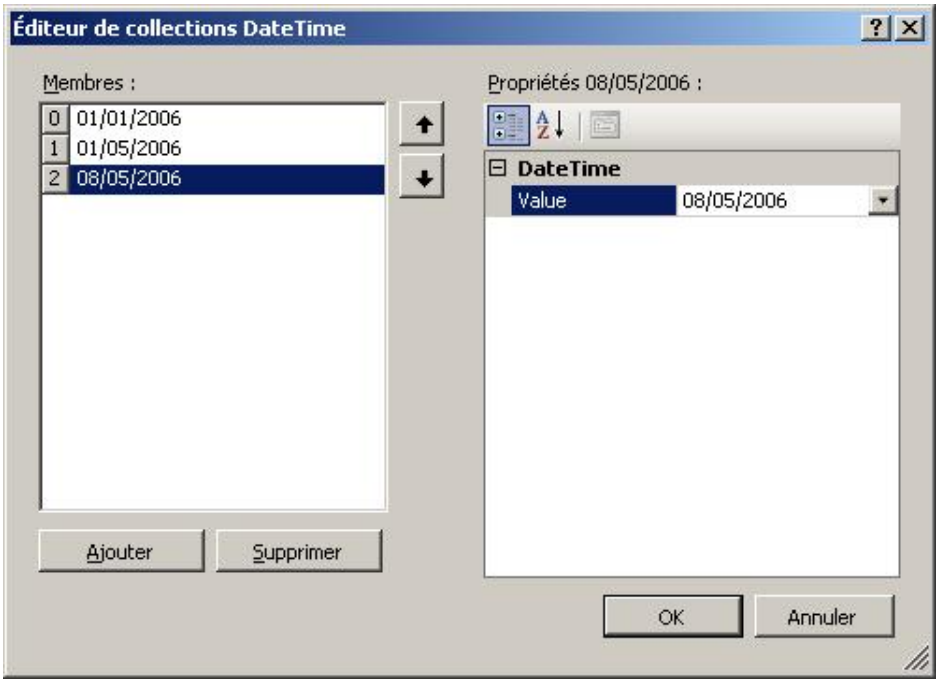
a. Le contrôle DateTimePicker

Ce contrôle associe une zone de texte et un calendrier. La date sera soit saisie directement dans la zone de texte (avec quand même une vérification des informations saisies), soit beaucoup plus facilement en affichant le calendrier en utilisant le bouton situé à coté de la zone de texte (comme pour une ComboBox).

La propriété `Value` permet de récupérer la date saisie ou au contraire de spécifier la date sélectionnée sur le calendrier à son affichage. Par défaut, le calendrier est initialisé avec la date du jour.

b. Le contrôle MonthCalendar

Ce contrôle correspond, en fait, au calendrier associé au contrôle `DateTimePicker` avec bien sûr des fonctionnalités supplémentaires. Vous pouvez, par exemple, indiquer une liste de dates qui apparaîtront en gras sur le calendrier (par exemple les jours fériés). La propriété`AnnuallyBoldedDates` contient la liste de tous ces jours "spéciaux".



La propriété `MonthlyBoldedDates` permet d'indiquer les jours qui apparaîtront en gras chaque mois (le jour où vous avez une facture à payer par exemple !).

L'intérêt principal de ce contrôle est qu'il permet la sélection d'une plage de dates. La propriété `MaxSelectionCount` indique le nombre de jours maximal de la sélection (par défaut, 7). Il reste un problème cependant, car il n'y a pas de solution simple pour choisir une plage de date "à cheval" sur plusieurs mois. Les calendriers des mois compris dans la sélection doivent être tous visibles, pour permettre la sélection d'une plage sur plusieurs mois. La solution consiste donc à configurer le contrôle pour qu'il affiche plusieurs mois simultanément. La propriété `CalendarDimensions` fixe le format du calendrier.

Par exemple, une application utilisée pour gérer un planning de vacances pourrait utiliser la configuration suivante :

```
      ' indique les jours fériés pour affichage en gras
        Me.Planning.AnnuallyBoldedDates = New Date() {New Date(2006, 7, 14, 0,
0, 0, 0), New Date(2006, 8, 15, 0, 0, 0, 0)}
        'affichage de trois mois en largeur sur un mois en hauteur
        Me.Planning.CalendarDimensions = New System.Drawing.Size(3, 1)
        Me.Planning.Location = New System.Drawing.Point(40, 128)
        'limite les possiblités de déplacement  du 1 Juin au 31 Aout 2002
        Me.Planning.MinDate = New Date(2006, 6, 1, 0, 0, 0, 0)
        Me.Planning.MaxDate = New Date(2006, 8, 31, 0, 0, 0, 0)
        'Autorise la sélection d'au maximum 28 jours (4 semaines de congés)
        Me.Planning.MaxSelectionCount = 28
        Me.Planning.Name = "Planning"
        Me.Planning.TabIndex = 6
        Me.Planning.TodayDate = New Date(2006, 6, 4, 0, 0, 0, 0)
```

Vous pouvez, ensuite, obtenir l'intervalle sélectionné en utilisant la propriété `SelectionRange` qui contient elle-même une propriété `Start` et une propriété `End`. L'événement `DateSelected` se déclenche lorsque l'une des bornes de la sélection est modifiée. On peut donc l'utiliser pour mettre à jour une zone de texte, comme dans l'exemple ci-dessous.

```
Private Sub Planning_DateSelected(ByVal sender As Object, ByVal e As
```

```
System.Windows.Forms.DateRangeEventArgs) Handles Planning.DateSelected
TxtVacances.Text = "vous êtes en vacances du " &    Planning.Selection
Range.Start & " au " & Planing.SelectionRange.End
End Sub
```

c. Le contrôle Timer

Ce contrôle va nous permettre de déclencher des événements à intervalles réguliers dans l'application. Ce contrôle est très simple d'utilisation puisqu'il ne possède que deux propriétés.

`Interval`

Indique le délai entre deux événements. Ce délai est exprimé en millisecondes.

`Enabled`

Indique si le contrôle génère des événements ou est inactif.

À chaque expiration du délai, un événement `Tick` est déclenché.

Il faut cependant être prudent lors de l'utilisation de ce contrôle en tenant compte des remarques suivantes :

- Si le système est très chargé (parce qu'il effectue des opérations d'entrée/sortie réseau par exemple), il se peut que les événements `Tick` ne soient pas déclenchés réguliè- rement.

- La précision du contrôle `Timer` n'est pas digne d'une montre Suisse. Il ne faut pas, par exemple, incrémenter une variable sur l'événement `Tick` d'un contrôle `Timer` pour mesurer une durée, mais plutôt se servir de l'horloge système comme dans l'exemple ci-dessous :

```
Dim depart As Date

Private Sub Form16_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles MyBase.Load
        depart = Now
End Sub

Private Sub Timer1_Tick(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles Timer1.Tick
  Dim durée As Double
‘ calcul de la durée heure courante-heure depart et conversion ‘en jour
  durée = DateDiff(DateInterval.Second, depart, Now) / (24 * 3600)
‘ conversion de la valeur en date et formatage en minutes : secondes
  lblDurée.Text = Format(Date.FromOADate(durée), "’vous travaillez  depuis’
mm ’minutes’ ss ’secondes’")
End Sub
```

d. Le composant BackGroundWorker

Il arrive parfois d'avoir à effectuer, dans une application, des opérations relativement longues comme par exemple des chargements d'images ou des recherches sur un disque dur. Ces opérations entraînent un blocage de l'application pendant leur exécution. L'utilisateur a l'impression que l'application ne répond plus et il est parfois tenté de mettre fin à l'application brutalement. Une solution envisageable est de fournir à l'utilisateur une information sur l'avancement du traitement avec par exemple l'affichage d'une barre de progression. Même avec cette solution l'utilisateur sera obligé d'attendre la fin du traitement pour pouvoir continuer à utiliser l'application. Le composant `BackGroundWorker` permet de résoudre ce problème en exécutant ces opérations de façon asynchrone en arrière-plan. Le code est exécuté sur un thread différent du thread principal de l'application qui lui continuera à gérer l'interface de l'application. Vous devez juste indiquer au `BackGroundWorker` le code qu'il doit exécuter puis appeler sa méthode `RunWorkerAsync`. À la fin de l'exécution, celui-ci vous préviendra en déclenchant l'événement `RunWorkerCompleted`. Nous allons tester ceci en réalisant une application calculant combien il y a de nombres premiers entre 0 et une valeur fixée. Tout d'abord voici le code d'une fonction vérifiant si un nombre est premier.

```
Public Function estPremier(ByVal nb As Integer) As Boolean
        If nb < 2 Then
            Return False
```

```
        End If
        If nb = 2 Then
            Return True
        End If
        If nb Mod 2 = 0 Then
            Return False
        End If
        Dim i As Integer
        i = 3
        Do While (i * i <= nb)
            If nb Mod i = 0 Then
                Return False
            Else
                i = i + 1
            End If
        Loop
        Return True
    End Function
```

Ce code est loin d'être le plus efficace en temps de calcul pour réaliser cette opération mais c'est un petit peu le but recherché. La deuxième fonction va calculer combien il y a de nombres premiers entre 0 et la valeur passée comme paramètre.

```
Public Function comptePremier(ByVal maxi As Integer)
        Dim i, nb As Integer
        For i = 0 To maxi
            If estPremier(i) Then
                nb = nb + 1
            End If
        Next
        Return nb
End Function
```

Nous devons ensuite placer sur le formulaire un contrôle `BackGroundWorker`, deux `TextBox` et deux `Button`. La première zone de texte servira pour la saisie de la valeur maximum de calcul, la deuxième pour l'affichage du résultat. Le premier bouton lancera le calcul par l'intermédiaire du `BackGroundWorker`, le deuxième lancera le calcul directement. Pour vérifier la réactivité de l'application, nous ajoutons également un contrôle `TextBox` multiligne par exemple.

Écrivons maintenant les gestionnaires d'événements pour les deux boutons. Le premier est simple puisque nous appelons simplement la fonction `comptePremier` en lui passant la valeur saisie dans la zone de texte `txtNbCalculs` et nous affichons le résultat dans la zone de texte `txtResultat`. La modification des propriétés `Enabled` des boutons évite de lancer une deuxième fois le calcul avant que le précédent soit terminé.

```
Private Sub cmdNormal_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles cmdNormal.Click

        cmdNormal.Enabled = False
        cmdBackGround.Enabled = False
        txtResultat.Text = comptePremier(txtNbCalculs.Text)
        cmdNormal.Enabled = True
        cmdBackGround.Enabled = True

End Sub
```

Le code du bouton `BackGround` est aussi simple. La seule grosse différence se situe au niveau de l'appel de la fonction `comptePremier` qui est en fait appelée indirectement par la méthode `RunWorkerAsync`. Le paramètre passé à cette méthode sera ensuite envoyé à la fonction `comptePremier`.

```
Private Sub cmdBackGround_click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles cmdBackGround.Click

        cmdBackGround.Enabled = False
        cmdNormal.Enabled = False
        txtResultat.Text = ""
        bgw.RunWorkerAsync(txtNbCalculs.Text)

End Sub
```

Il nous reste encore deux choses à préciser pour que l'application soit opérationnelle. Nous devons indiquer au `BackGroundWorker` quel code il doit exécuter en tâche de fond. Pour cela il faut gérer son événement `DoWork` qui lui, est

exécuté sur le thread associé au BackGroundWorker. L'information à fournir à la fonction comptePremier est obtenue par l'intermédiaire de la propriété Argument du paramètre.

```
Private Sub bgw_DoWork(ByVal sender As Object, ByVal e
As System.ComponentModel.DoWorkEventArgs) Handles bgw.DoWork

        e.Result = comptePremier(e.Argument)

End Sub
```

Le résultat doit être transmis dans la propriété Result. Le BackGroundWorker nous prévient qu'il a terminé le travail en déclenchant l'événement RunWorkerCompleted. En réponse à cet événement, nous récupérons le résultat par l'intermédiaire de la propriété Result et nous l'affichons dans la zone de texte de résultat.

```
Private Sub bgw_RunWorkerCompleted(ByVal sender As Object, ByVal e As
System.ComponentModel.RunWorkerCompletedEventArgs) Handles bgw.RunWorkerCompleted

        txtResultat.Text = e.Result
        cmdBackGround.Enabled = True
        cmdNormal.Enabled = True

End Sub
```

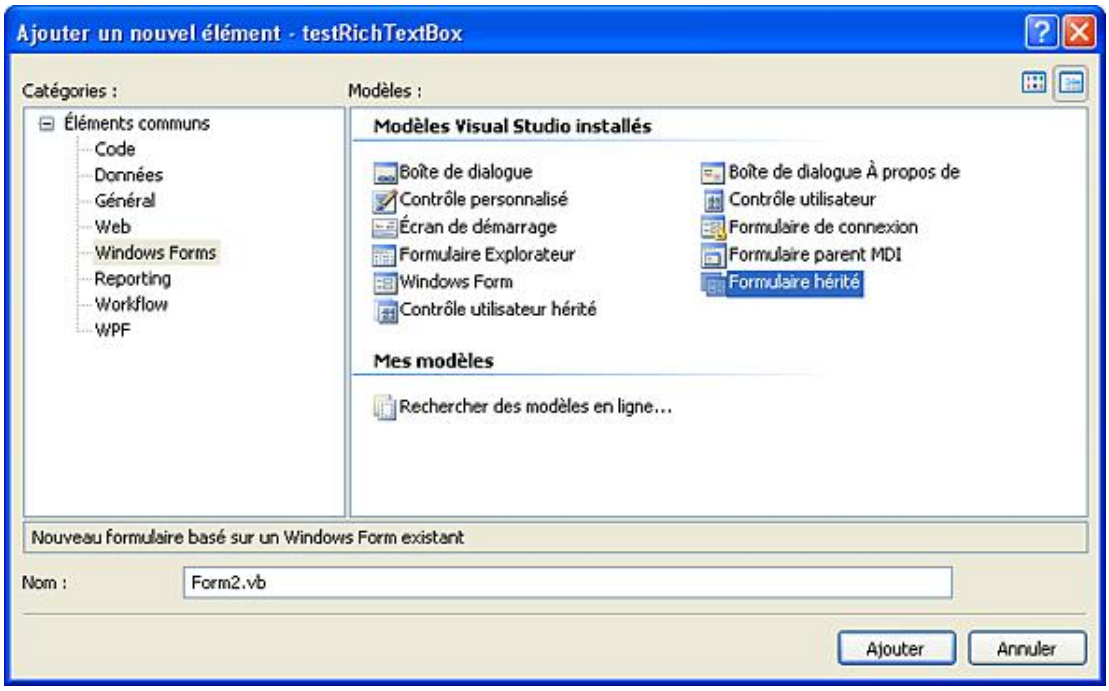
Il y a juste un piège à éviter dans l'utilisation du BackGroundWorker. Le code de l'événement DoWork ne doit absolument pas accéder aux autres contrôles du formulaire car ils ne sont pas gérés par le même Thread. Si vous tentez de le faire, vous obtiendrez l'exception suivante.

Pour tester le fonctionnement, vous devez simplement fournir une valeur (environ 10000000) puis lancer le calcul par l'un des deux boutons. Pendant que le calcul s'exécute vous pouvez essayer d'utiliser le troisième contrôle TextBox. Dans le cas du traitement par le BackGroundWorker, ce contrôle est utilisable en cours de calcul. L'appel direct de la fonction comptePremier bloque l'interface de l'application pendant son exécution.

L'héritage de formulaires

Vous pouvez parfois avoir besoin qu'un projet appelle un formulaire similaire à un autre que vous avez déjà créé dans un autre projet. Vous pouvez également créer un formulaire de base contenant des paramètres tel qu'un arrière-plan statique ou une présentation particulière des contrôles que vous comptez réutiliser plusieurs fois dans un projet, chaque nouvelle version contenant des modifications par rapport au modèle d'origine. L'héritage de formulaire vous permet de créer un formulaire de base puis d'en hériter pour personnaliser les nouvelles versions ainsi créées.

Pour pouvoir créer un formulaire hérité, il faut bien sûr au préalable concevoir le formulaire de base. Pour que l'héritage de formulaire soit accessible, le projet contenant le formulaire de base doit obligatoirement avoir été compilé. L'ajout d'un formulaire hérité est réalisable par l'utilisation de la boîte de dialogue classique d'ajout d'élément à un projet en choisissant l'option **Formulaire hérité**.



Nommez ensuite votre nouveau formulaire et cliquez sur le bouton **Ajouter**. La boîte de dialogue **Sélecteur d'héritage** s'ouvre et si le projet actuel contient déjà des formulaires, ils sont affichés dans cette boîte de dialogue. Pour hériter d'un formulaire disponible dans un autre assembly, cliquez sur le bouton **Parcourir** et sélectionnez le fichier (.exe ou .dll) contenant le formulaire de base puis validez votre choix avec le bouton **OK**. Le nouveau formulaire est alors ajouté à votre projet. Sur ce formulaire, les contrôles provenant de l'héritage sont marqués grâce au symbole  .

La propriété `Modifiers` de chacun des contrôles du formulaire de base détermine les actions possibles sur ces contrôles dans un formulaire hérité. Les règles standard de l'héritage sont appliquées. Le tableau suivant résume ces règles de visibilité.

- Public : les contrôles peuvent être redimensionnés et déplacés. Le contrôle est accessible en interne par la classe qui le déclare et en externe par les autres classes.

- Protected : les contrôles peuvent être redimensionnés et déplacés. Le contrôle est accessible en interne par la classe qui le déclare et par toute classe héritant de la classe parente, mais il n'est pas accessible aux classes externes.

- Protected Friend : les contrôles peuvent être redimensionnés et déplacés. Peuvent être accessibles en interne par la classe qui les déclare, par toute classe qui hérite de la classe parente, et par d'autres membres de l'assembly qui les contient.

- Friend : tous les aspects du contrôle sont considérés comme accessibles seulement en lecture. Vous ne pouvez pas le déplacer ou le redimensionner, ni modifier ses propriétés. Le contrôle est accessible uniquement par d'autres membres de l'assembly qui le contient.

- Private : tous les aspects du contrôle sont considérés comme accessibles seulement en lecture. Vous ne pouvez pas le déplacer ou le redimensionner, ni modifier ses propriétés. Le contrôle n'est accessible que depuis la classe qui le déclare.

D’autres contrôles peuvent bien sûr être ajoutés sur le formulaire hérité pour personnaliser son aspect. Si le formulaire de base est modifié après son utilisation dans une relation d’héritage, les modifications sont propagées aux formulaires hérités lors de la compilation du formulaire de base.

Principe de fonctionnement d'une base de données

Les bases de données sont devenues des éléments incontournables de la majorité des applications. Elles se substituent à l'utilisation de fichiers gérés par le développeur lui-même. Cet apport permet un gain de productivité important lors du développement et une amélioration significative des performances des applications. Elles facilitent également le partage d'informations entre utilisateurs. Pour pouvoir utiliser une base de données, vous devez connaître un minimum de vocabulaire lié à cette technologie.

1. Terminologie

Dans le contexte des bases de données, les termes suivants sont fréquemment utilisés :

Base de données relationnelle

Une base de donnée relationnelle est un type de base de données qui utilise des tables pour le stockage des informations. Elles utilisent des valeurs issues de deux tables, pour associer les données d'une table aux données d'une autre table. En règle générale, dans une base de données relationnelle, les informations ne sont stockées qu'une seule fois.

Table

Une table est un composant d'une base de données qui stocke les informations dans des enregistrements (lignes) et dans des champs (colonnes). Les informations sont, en général, regroupées par catégorie au niveau d'une table. Par exemple, nous aurons la table des Clients, des Produits ou des commandes.

Enregistrement

L'enregistrement est l'ensemble des informations relatives à un élément d'une table. Les enregistrements sont les équivalents, au niveau logique, des lignes d'une table. Par exemple, un enregistrement de la table **Clients** contient les caractéristiques d'un client particulier.

Champ

Un enregistrement est composé de plusieurs champs. Chaque champ d'un enregistrement contient une seule information sur l'enregistrement. Par exemple, un enregistrement Client peut contenir les champs **CodeClient**, **Nom**, **Prénom**...

Clé primaire

Une clé primaire est utilisée pour identifier, de manière unique, chaque ligne d'une table. La clé primaire est un champ ou une combinaison de champs dont la valeur est unique dans la table. Par exemple, le champ **CodeClient** est la clé primaire de la table **Client**. Il ne peut pas y avoir deux clients ayant le même code.

Clé étrangère

Une clé étrangère représente un ou plusieurs champs d'une table, qui font référence aux champs de la clé primaire d'une autre table. Les clés étrangères indiquent la manière dont les tables sont liées.

Relation

Une relation est une association établie entre des champs communs dans deux tables. Une relation peut être de un à un, de un à plusieurs ou de plusieurs à plusieurs. Grâce aux relations, les résultats de requêtes peuvent contenir des données issues de plusieurs tables. Une relation de un à plusieurs entre la table **Client** et la table **Commande** permet à une requête de renvoyer toutes les commandes correspondant à un client.

2. Le langage SQL

Avant de pouvoir écrire une application Visual Basic utilisant des données, vous devez être familiarisé avec le langage SQL (*Structured Query Language*). Ce langage permet de dialoguer avec la base de données. Il existe différentes versions du langage SQL, en fonction de la base de données utilisée. Cependant, SQL dispose également d'une syntaxe élémentaire, normalisée, indépendante de toutes bases de données.

a. Recherche d'informations

Le langage SQL permet de spécifier les enregistrements à extraire ainsi que l'ordre dans lequel vous souhaitez les extraire. Vous pouvez créer une instruction SQL qui extrait des informations de plusieurs tables simultanément, ou créer une instruction qui extrait uniquement un enregistrement spécifique.

L'instruction`SELECT` est utilisée pour renvoyer des champs spécifiques d'une ou de plusieurs tables de la base de données.

L'instruction suivante renvoie la liste des noms et prénoms de tous les enregistrements de la table **Client** :

```
SELECT Nom,Prenom FROM Client
```

Vous pouvez utiliser le symbole * à la place de la liste des champs pour lesquels vous souhaitez la valeur :

```
SELECT * FROM Client
```

Vous pouvez limiter le nombre d'enregistrements sélectionnés, en utilisant un ou plusieurs champs pour filtrer le résultat de la requête. Différentes clauses sont disponibles pour exécuter ce filtrage.

Clause WHERE

Cette clause permet de spécifier la liste des conditions, que devront remplir les enregistrements pour faire partie des résultats retournés. L'exemple suivant permet de retrouver tous les clients habitant Nantes :

```
SELECT * FROM Client WHERE Ville='Nantes'
```

La syntaxe de cette clause nécessite l'utilisation de simple cote pour la délimitation des chaînes de caractères.

Clause WHERE ... IN

Vous pouvez utiliser la clause `WHERE ... IN` pour renvoyer tous les enregistrements qui répondent à une liste de critères. Par exemple, vous pouvez rechercher tous les clients habitant en France ou en Espagne :

```
SELECT * FROM Client WHERE Pays IN ('France','Espagne')
```

Clause WHERE ... BETWEEN

Vous pouvez également renvoyer une sélection d'enregistrements qui se situent entre deux critères spécifiés. La requête suivante permet de récupérer la liste des commandes passées au mois de novembre 2005 :

```
SELECT * from Commandes WHERE DateCommande BETWEEN '01/11/05' AND '30/11/05'
```

Clause WHERE ... LIKE

Vous pouvez utiliser la clause `WHERE ... LIKE` pour renvoyer tous les enregistrements pour lesquels il existe une condition particulière pour un champ donné. Par exemple, la syntaxe suivante sélectionne tous les clients dont le nom commence par un d :

```
SELECT * FROM Client WHERE Nom LIKE 'd%'
```

Dans cette instruction, le symbole % est utilisé pour remplacer une séquence de caractères quelconque.

Clause ORDER BY ...

Vous pouvez utiliser la clause `ORDER BY` pour renvoyer les enregistrements dans un ordre particulier. L'option `ASC` indique un ordre croissant, l'option `DESC` indique un ordre décroissant. Plusieurs champs peuvent être spécifiés comme critère de tri. Ils sont analysés de la gauche vers la droite. En cas d'égalité sur la valeur d'un champ, le champ suivant est utilisé :

```
SELECT * FROM Client ORDER BY Nom DESC,Prenom ASC
```

Cette instruction retourne les clients triés par ordre décroissant sur le nom et, en cas d'égalité, par ordre croissant sur le prénom.

b. Ajout d'informations

La création d'enregistrements dans une table s'effectue par la commande `INSERT INTO`. Vous devez indiquer la table dans laquelle vous souhaitez insérer une ligne, la liste des champs pour lesquels vous spécifiez une valeur et, enfin, la liste des valeurs correspondantes. La syntaxe complète est donc la suivante :

```
INSERT INTO client (codeClient,nom,prenom) VALUES (1000,'Dupond','Pierre')
```

Lors de l'ajout de ce nouveau client, seuls le nom et le prénom seront renseignés dans la table. Les autres champs prendront la valeur **NULL**. Si la liste des champs n'est pas indiquée, l'instruction `insert` exige que vous spécifiez une valeur pour chaque champ de la table. Vous êtes donc obligés d'utiliser le mot clé **NULL** pour indiquer que, pour un champ particulier, il n'y a pas d'information. Si la table **Client** est composée de cinq champs (codeClient,nom,prenom,adresse,pays), l'instruction précédente peut être écrite avec la syntaxe suivante :

```
INSERT INTO client VALUES (1000,'Dupond','Pierre',NULL,NULL)
```

Dans ce cas, les deux mots clés NULL sont obligatoires pour les champs adresse et pays.

c. Mise à jour d'informations

La modification des champs pour des enregistrements existants, s'effectue par l'instruction`UPDATE`. Cette instruction peut mettre à jour plusieurs champs de plusieurs enregistrements d'une table, à partir des expressions qui lui sont fournies. Pour cela, vous devez fournir le nom de la table à mettre à jour ainsi que la valeur à affecter aux différents champs. La liste est indiquée par le mot clé **SET** suivi de l'affectation de la nouvelle valeur aux différents champs. Si vous voulez que les modifications ne portent que sur un ensemble limité d'enregistrements, vous devez spécifier la clause `WHERE`, afin de limiter la portée de la mise à jour. Si aucune clause `WHERE` n'est indiquée, la modification se fera sur l'ensemble des enregistrements de la table.

Par exemple, pour modifier l'adresse d'un client particulier, vous pouvez utiliser l'instruction suivante :

```
UPDATE Client SET adresse= '4 rue de Paris 44000 Nantes' WHERE codeClient=1000
```

Si la modification porte sur l'ensemble des enregistrements de la table, la clause `WHERE` est inutile. Par exemple, si vous souhaitez augmenter le prix unitaire de tous vos articles, vous pouvez utiliser l'instruction suivante :

```
UPDATE CATALOGUE SET prixUnitaire=prixUnitaire*1.1
```

d. Suppression d'informations

L'instruction`DELETE FROM` permet de supprimer un ou plusieurs enregistrements d'une table. Vous devez, au minimum, fournir le nom de la table sur laquelle va s'effectuer la suppression. Si vous n'indiquez pas plus de précisions, toutes les lignes de la table sont supprimées. En général, une clause `WHERE` est ajoutée pour limiter l'étendue de la suppression. La commande suivante efface tous les enregistrements de la table **Client** :

```
DELETE FROM Client
```

La commande suivante est moins radicale et ne supprime qu'un enregistrement particulier :

```
DELETE FROM Client WHERE codeClient=1000
```

Le langage SQL est, bien sûr, beaucoup plus complet que cela et ne se résume pas à ces cinq instructions. Néanmoins, elles sont suffisantes pour la manipulation de données à partir de Visual Basic. Si vous souhaitez approfondir l'apprentissage du langage SQL, consultez un des ouvrages disponibles dans la même collection traitant de ce sujet de manière plus poussée.

Présentation d'ADO.NET

ADO.NET est un ensemble de classes, d'interfaces, de structures et d'énumérations permettant la manipulation des données. Les différents composants d'ADO.NET permettent de séparer l'accès aux données de la manipulation des données. ADO.NET facilite également l'utilisation du langage XML, en permettant la conversion de données relationnelles au format XML ou l'importation de données aux formats XML dans un modèle relationnel. Deux modes de fonctionnement sont disponibles dans ADO.NET :

- le mode connecté ;
- le mode non connecté.

1. Mode connecté

Dans un environnement connecté, l'application ou l'utilisateur est en permanence connecté à la source de données. Depuis les débuts de l'informatique, c'était le seul mode disponible. Ce mode présente certains avantages dans son fonctionnement :

- Il est facile à gérer : la connexion est réalisée au début de l'application puis est coupée à sa fermeture.
- L'accès concurrentiel est plus facile à contrôler : comme tous les utilisateurs sont connectés en permanence, il est plus facile de contrôler lequel travaille sur les données.
- Les données sont à jour : toujours grâce à la connexion permanente aux données, il est facilement envisageable de prévenir toutes les applications utilisant les données que des modifications viennent d'y être apportées.

Par contre, certains inconvénients viennent un peu noircir le tableau :

- La connexion réseau doit être constamment maintenue : en cas d'utilisation de l'appli- cation sur un ordinateur portable, l'accès au réseau risque de ne pas être disponible en permanence.
- Il y a un risque de gaspillage de ressources sur le serveur : au moment de l'établis- sement d'une connexion entre une application cliente et un serveur, des ressources sont allouées sur le serveur pour la gestion de cette connexion. Ces ressources restent monopolisées par la connexion, même si aucune information ne transite par cette connexion.

Cependant, dans certaines situations, l'utilisation d'un mode connecté est incontournable. C'est le cas, par exemple, des applications effectuant des traitements en temps réel.

2. Mode non connecté

Un mode non connecté signifie qu'une application ou un utilisateur n'est pas constamment connecté à une source de données. Les applications Internet utilisent souvent ce mode de fonctionnement. La connexion aux données est ouverte, les données sont extraites puis la connexion est coupée. L'utilisateur travaille avec les données, à partir de son navigateur, et la connexion est à nouveau ouverte pour la mise à jour de la source de données ou l'obtention d'autres données. Les utilisateurs, travaillant sur des ordinateurs portables, sont également les principaux utilisateurs d'environnements déconnectés. Un médecin peut, par exemple le matin, charger les dossiers médicaux des patients qu'il va visiter dans la journée, puis le soir, fusionner les modifications dans la base de données. Les avantages d'un environnement déconnecté sont les suivants :

- Les connexions sont utilisées pendant la plus courte durée possible. De cette façon, un petit nombre de connexions disponibles sur un serveur suffisent pour de nombreux utilisateurs.
- Un environnement déconnecté améliore l'évolutivité et les performances d'une appli- cation, en optimisant la disponibilité des connexions.

L'environnement déconnecté comporte cependant quelques inconvénients :

- Les données disponibles dans l'application ne sont pas toujours à jour. Par exemple, dans le cas de notre médecin, si sa secrétaire ajoute des résultats d'analyse après qu'il ait récupéré les dossiers médicaux de ces patients, il ne pourra pas disposer immé- diatement des informations.
- Des conflits peuvent parfois survenir lors de la mise à jour des informations dans la base. Ce type de problèmes doit être pris en charge lors de la conception de l'application. Différentes approches sont disponibles pour la gestion de ces conflits :
    - Autoriser la prédominance des mises à jour, les plus récentes, en écrasant les données déjà présentes dans la base.
    - Autoriser la prédominance des mises à jour, les plus anciennes, en abandonnant les nouvelles mises à jour.
    - Prévoir du code permettant à l'utilisateur de choisir ce qu'il souhaite faire en cas de conflit lors d'une mise à jour.

3. Architecture d'ADO.NET

Le but d'ADO.NET est de fournir un ensemble de classes permettant l'accès aux bases de données. Deux types de composants sont disponibles :

- Les fournisseurs de données, spécifiques à un type de base de données. Ils assurent la communication avec un type spécifique de base de données et permettent la manipulation des données directement dans la base en mode connecté. Les possibilités sont cependant limitées puisque uniquement un accès en lecture seule est disponible.

- Les classes de manipulation des données, indépendantes du type de base de données, voire utilisables sans base de données, permettent la manipulation locale des données dans l'application.

4. Les fournisseurs de données

Les fournisseurs de données servent de passerelle entre une application et une base de données. Ils sont utilisés pour récupérer les informations, à partir de la base de données, et transférer les changements effectués sur les données par l'application vers la base de données. Quatre fournisseurs de données sont disponibles dans le Framework.NET :

- le fournisseur pour SQL Server ;

- le fournisseur pour OLE DB ;

- le fournisseur pour ODBC ;

- le fournisseur pour Oracle.

Ils proposent tous l'implémentation de quatre classes, de base, nécessaires pour le dialogue avec la base de données :

- La classe **Connection** permet d'établir une connexion avec le serveur de base de données.

- La classe **Command** permet de demander l'exécution d'une instruction ou d'un ensemble d'instructions SQL à un serveur.

- La classe **DataReader** procure un accès en lecture seule et un défilement, en avant seulement, aux données, (même principe qu'un fichier séquenciel).

- La classe **DataAdapter** est utilisée pour assurer le transfert des données vers un système de cache local à l'application (le `DataSet`) et mettre à jour la base de données, en fonction des modifications effectuées localement dans le `DataSet`.

Quelques autres classes sont disponibles pour, par exemple, la gestion des transactions, ou le passage de paramètres à une instruction SQL.

a. SQL Server

Le fournisseur de données pour SQL Server utilise un protocole natif pour dialoguer avec le serveur de base de données. Il est également peu consommateur de ressources puisqu'il accède au serveur, sans utiliser de couche logicielle supplémentaire telle que OLE DB ou ODBC. Il est utilisable avec SQL Server à partir de la version 7. Toutes les classes de ce fournisseur de données sont disponibles dans l'espace de nom **System.Data.SqlClient**. Dans cet espace de nom, le nom de chaque classe est préfixé par `Sql`. Ainsi, la classe permettant de se connecter à un serveur SQL Server s'appelle `SqlConnection`.

b. OLE DB

Le fournisseur OLE DB utilise la couche logicielle OLE DB pour communiquer avec le serveur de base de données. Vous pouvez utiliser ce fournisseur pour dialoguer avec une base de données pour laquelle il n'existe pas de fournisseur spécifique, mais pour laquelle le pilote OLE DB est disponible. Avec cette solution, le fournisseur ne contacte pas le serveur directement mais passe par le pilote OLE DB pour communiquer. Pour que cette communication soit possible, le pilote doit implémenter certaines interfaces. Toutes les classes sont disponibles dans l'espace de nom **System.Data.OleDb**. Les noms de classe de cet espace de nom sont préfixés par `OleDb`. Pour pouvoir fonctionner correctement, ce fournisseur exige l'installation de MDAC 2.6 sur la machine (*Microsoft Data Access Components*).

c. ODBC

Le fournisseur ODBC utilise un pilote ODBC natif pour communiquer avec le serveur de base de données. Ce fournisseur utilise un pilote ODBC natif pour la communication. Le principe est identique à celui utilisé pour le fournisseur OLE DB. Toutes les classes sont disponibles dans l'espace de nom **System.Data.Odbc**. Les noms de classes sont préfixés par `Odbc`. Pour pouvoir fonctionner correctement, ce fournisseur exige l'installation sur la machine de MDAC 2.6 (*Microsoft Data Access Components*).

d. ORACLE

Le fournisseur pour Oracle permet la connexion à une source de données Oracle, à travers les outils client Oracle. Ces outils client doivent être installés sur le système pour pouvoir se connecter à une base Oracle. La version 8.1.7 ou supérieure est exigée. Les classes sont localisées dans l'espace de nom **System.Data.OracleClient** et utilisent Oracle comme préfixe de nom. Pour utiliser le fournisseur pour Oracle, vous devrez également ajouter une référence vers la bibliothèque **System.Data.OracleClient.dll**.

5. Rechercher les fournisseurs disponibles

Pour assurer le bon fonctionnement d'une application utilisant un accès aux données, les fournisseurs de données doivent être disponibles sur le poste client. La classe `DbProviderFactories` propose la méthode partagée `GetFactoryClasses`, permettant d'énumérer les fournisseurs de données disponibles sur le poste. L'exemple de code suivant affiche le nom, la description et l'espace de nom racine de chacun des fournisseurs installés sur le poste de travail.

```
Imports System.Data
Imports System.Data.Common
Module ListeProviders
    Sub Main()
        Dim resultat As DataTable
        'récupération de la liste des fournisseurs dans une dataTable
        resultat = DbProviderFactories.GetFactoryClasses()
        Dim colonne As DataColumn
        Dim ligne As DataRow
        'parcours des colonnes de la dataTable et affichage du nom
        For Each colonne In resultat.Columns
            Console.Write(colonne.ColumnName & vbTab)
        Next
        Console.WriteLine()
        ' parcours de la dataTable et affichage de chaque ligne
        For Each ligne In resultat.Rows
            ' parcours de chaque ligne et affichage de chaque champ
            For Each colonne In resultat.Columns
                Console.Write(ligne(colonne.ColumnName) & vbTab)
            Next
            Console.WriteLine()
        Next
        Console.ReadLine()
        Stop
    End Sub
End Module
```

6. Compatibilité du code

En fonction du fournisseur utilisé, vous devez importer l'espace de nom correspondant pour avoir un accès facile aux classes du fournisseur. Cependant, comme les classes de chacun des fournisseurs ne portent pas le même nom, votre code sera spécifique à un type de fournisseur. Il est toutefois possible d'écrire du code pratiquement indépendant du type de fournisseur. Pour cela, au lieu d'utiliser les classes spécifiques à chacun des fournisseurs, vous pouvez utiliser comme type de données les interfaces qu'elles implémentent. L'utilisation d'une classe spécifique n'est indispensable que pour la création de la connexion. Une fois que la connexion est créée, vous pouvez travailler uniquement avec des interfaces. L'exemple de code suivant liste le contenu d'une table d'une base SQL Server, en utilisant uniquement des interfaces.

```
Module accesBdParInterfaces
    Dim ctn As IDbConnection
    Public Sub main()
        ' c'est la seule ligne de code spécifique à un fournisseur
        ctn = New System.Data.SqlClient.SqlConnection("Data Source=TG;Initial
Catalog=Northwind;Integrated Security=True")
        Dim cmd As IDbCommand
        cmd = ctn.CreateCommand
        ctn.Open()
        cmd.CommandText = "select * from products"
        Dim lecteur As IDataReader
        lecteur = cmd.ExecuteReader
       Console.WriteLine("Lecture des données dans une base SQL Server")
       Do While lecteur.Read
            Console.WriteLine("numéro : {0} nom produit : {1}",
lecteur.GetInt32(0), lecteur.GetString(1))
        Loop
    End Sub
End Module
```

L'exécution de ce code affiche le résultat suivant :

```
Lecture des données dans une base SQL Server
numéro : 56 nom produit : Gnocchi di nonna Alice
numéro : 57 nom produit : Ravioli Angelo
numéro : 58 nom produit : Escargots de Bourgogne
numéro : 59 nom produit : Raclette Courdavault
numéro : 60 nom produit : Camembert Pierrot
numéro : 61 nom produit : Sirop d'erable
numéro : 62 nom produit : Tarte au sucre
```

Si cette application doit ensuite migrer vers un autre type de base de données, il n'y a que la ligne concernant la connexion à modifier. Si les données sont maintenant disponibles dans une base Access, la création de la connexion prend alors la forme suivante :

```
ctn = New System.Data.OleDb.OleDbConnection ("Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\Documents and Settings\tgroussard\Mes documents\livre vb.net 2005\ accès aux bases de données\exemples\NWIND.MDB")
```

L'exécution du code ainsi modifié génère bien le même résultat :

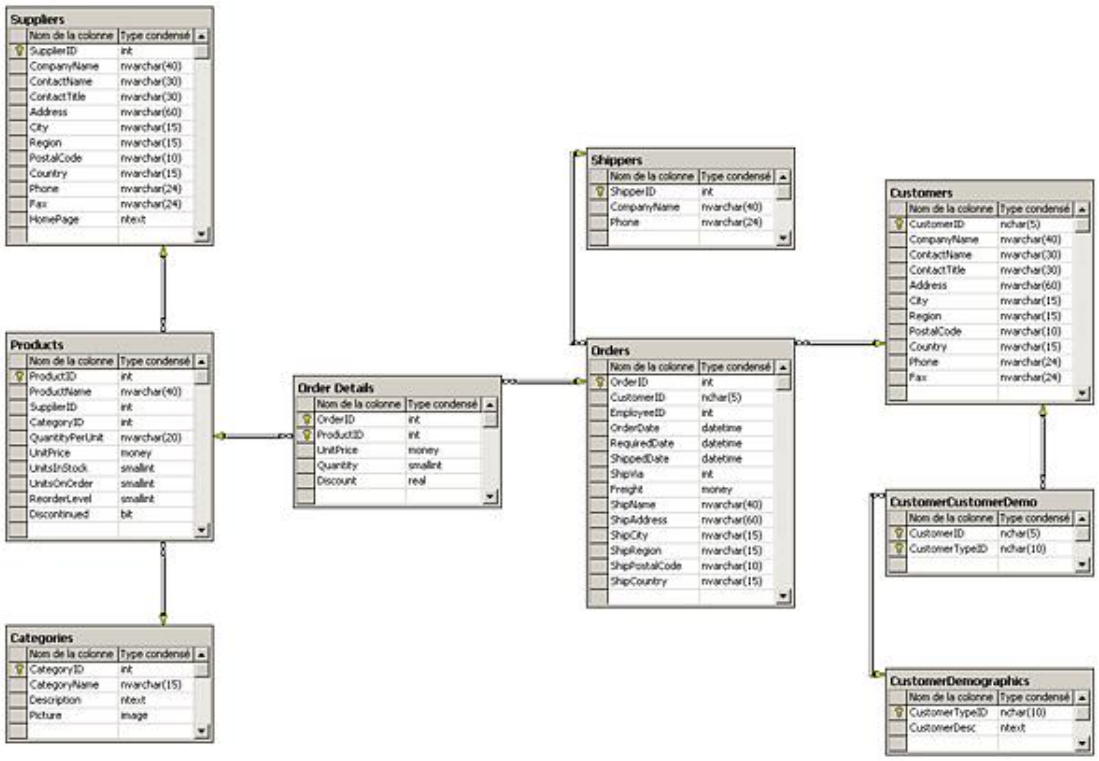
```
Lecture des données dans une base Access
numéro : 56 nom produit : Gnocchi di nonna Alice
numéro : 57 nom produit : Ravioli Angelo
numéro : 58 nom produit : Escargots de Bourgogne
numéro : 59 nom produit : Raclette Courdavault
numéro : 60 nom produit : Camembert Pierrot
numéro : 61 nom produit : Sirop d'erable
numéro : 62 nom produit : Tarte au sucre
```

Il convient, par contre, d'être prudent et de ne pas utiliser d'instructions SQL spécifiques à un type de base de données particulier. Pour faciliter la relecture du code, il est préférable de regrouper toutes les instructions SQL sous forme de constantes de type chaîne de caractères au début de chaque module. Avec cette technique, vous n'aurez pas à chercher des instructions SQL au milieu de centaines de lignes de code Visual Basic. Il convient d'être également prudent lors de l'utilisation de paramètres dans une instruction SQL. Le fournisseur pour SQL Server utilise des paramètres nommés donc l'ordre de création des paramètres n'a pas d'importance.

Le fournisseur pour OLE DB utilise la position des paramètres dans l'instruction SQL, pour le remplacement lors de l'exécution. L'ordre de la création des paramètres est donc, dans ce cas, capital pour le bon fonctionnement de l'instruction.

Utilisation du mode connecté

Dans ce chapitre, nous allons aborder les opérations pouvant être exécutées sur une base de données, en utilisant le mode connecté. Certaines notions étudiées dans ce chapitre seront également utiles pour le fonctionnement en mode déconnecté. Pour tester les différentes fonctionnalités étudiées dans ce chapitre, nous utiliserons un serveur SQL Server 2000. La base de données utilisée sera la base Northwind qui est créée par défaut à l'installation du serveur. Une partie de la structure de la base est disponible sur le schéma ci-dessous.



1. Connexion à une base

Pour pouvoir travailler avec un serveur de base de données, une application doit établir une connexion réseau avec le serveur. La classe `SqlConnection` est capable de gérer une connexion vers un serveur SQL Server version 7.0 ou ultérieure. Comme pour tout objet, nous devons en premier lieu déclarer une variable.

```
Dim ctn As System.Data.SqlClient.SqlConnection
```

Puis, nous devons créer l'instance de la classe et l'initialiser en appelant un constructeur. L'initialisation va consister essentiellement à indiquer les paramètres utilisés pour établir la connexion avec le serveur. Ces paramètres sont définis sous forme d'une chaîne de caractères. Ils peuvent être indiqués lors de l'appel du constructeur ou modifiés par la suite par la propriété `ConnectionString`.

a. Chaîne de connexion

Le format standard d'une chaîne de connexion est constitué d'une série de couples mot clé/ valeur séparés par des points virgules. Le signe = est utilisé pour l'affectation d'une valeur à un mot clé. L'analyse de la chaîne est effectuée lors de l'affectation de la chaîne à la propriété `ConnectionString`. Les valeurs associées aux mots clés sont alors extraites et affectées aux différentes propriétés de la connexion. Si une erreur de syntaxe est trouvée alors une exception est générée immédiatement et aucune propriété n'est modifiée. Par contre, certaines propriétés ne pourront être contrôlées que lors de l'ouverture de la connexion. C'est alors à ce moment qu'une exception sera déclenchée si la chaîne de connexion contient une erreur. La chaîne de connexion ne peut être modifiée que si la connexion est fermée. Les mots clés suivants sont disponibles pour une chaîne de connexion :

Connect Timeout

Durée en secondes pendant laquelle l'application attendra une réponse du serveur à sa demande de connexion. Passé ce délai, une exception est déclenchée.

Data Source

Nom ou adresse réseau du serveur vers lequel est établie la connexion. Le numéro du port peut être spécifié à la suite du nom ou de l'adresse réseau. S'il n'est pas indiqué, le numéro de port est égal à 1433.

Initial Catalog

Nom de la base sur laquelle doit s'effectuer la connexion.

Integrated Security

Si cette valeur est positionnée sur **false** alors un nom d'utilisateur et un mot de passe doivent être fournis dans la chaîne de connexion. Sinon, le compte Windows de l'utilisateur est utilisé pour l'authentification.

Persist Security Info

Si cette valeur est positionnée sur **true**, alors le nom de l'utilisateur et son mot de passe sont accessibles par la connexion. Pour des raisons de sécurité, cette valeur doit être positionnée sur **false**. C'est d'ailleurs le cas si vous n'indiquez rien dans votre chaîne de connexion.

Pwd

Mot de passe associé au compte SQL Server utilisé pour la connexion. S'il n'y a pas de mot de passe associé à un compte, cette information peut être omise dans la chaîne de connexion.

User ID

Nom du compte SQL Server utilisé pour la connexion.

Connection LifeTime

Indique la durée de vie d'une connexion dans un pool de connexions. Une valeur égale à zéro indique une durée de vie infinie.

Connection Reset

Indique si la connexion est réinitialisée lors de sa remise dans le pool.

Max Pool Size

Nombre maximum de connexions dans le pool.

Min Pool Size

Nombre minimum de connexions dans le pool.

Pooling

Indique si la connexion peut être extraite d'un pool de connexion.

Une chaîne de connexion prend donc la forme minimale suivante :

```
ctn.ConnectionString = "Data Source=Minerve;Initial Catalog=Northwind;Integrated
Security=true"
```

b. Pool de connexions

Les pools de connexions permettent d'améliorer les performances d'une application, en évitant la création de connexions supplémentaires. Lorsqu'une connexion est ouverte, un pool de connexions est créé en se basant sur un algorithme basé, lui-même sur la chaîne de connexion. Chaque pool est donc associé à une chaîne de connexion particulière. Si une nouvelle connexion est ouverte et qu'il n'existe pas de pool correspondant exactement à sa chaîne de connexion, alors un nouveau pool est créé. Les pools de connexions ainsi créés existeront jusqu'à la fin de l'application. Lors de la création du pool, d'autres connexions peuvent être créées automatiquement pour satisfaire la

valeur `Min Pool Size` indiquée dans la chaîne de connexion. D'autres connexions pourront, par la suite, être ajoutées au pool jusqu'à atteindre la valeur `Max Pool Size` de la chaîne de connexion. Lorsqu'une connexion est requise, elle peut être obtenue à partir d'un pool de connexion (s'il en existe un correspondant exactement aux caractéristiques de la connexion demandée). Il faut bien sûr que le pool en contienne une disponible et active.

Si le nombre maximum de connexion dans le pool est atteint, la demande est mise en file d'attente jusqu'à ce qu'une connexion soit à nouveau disponible. Une connexion est remise à la disposition du pool, lors de sa fermeture ou lors de l'appel de la méthode `Dispose` sur la connexion. Pour cette raison, il est recommandé de fermer explicitement les connexions lorsqu'elles ne sont plus utilisées dans l'application. Les connexions sont retirées du pool lorsque celui-ci détecte que la connexion n'a pas été utilisée depuis un certain temps, indiqué par la valeur `ConnectionLifeTime` de la chaîne de connexion. Elles sont également retirées du pool, s'il détecte que la connexion avec le serveur a été interrompue.

c. Événements de connexion

La classe `SQLConnection` propose deux événements vous permettant d'être prévenu lorsque l'état de la connexion change ou qu'un message d'information est envoyé par le serveur. L'événement `StateChanged` est déclenché lors d'un changement d'état de la connexion. Le gestionnaire de cet événement reçoit un paramètre de type `StateChangeEventArg` permettant d'obtenir, avec la propriété `CurrentState`, l'état actuel de la connexion et avec la propriété `OriginalState`, l'état de la connexion avant le déclenchement de l'événement. Pour tester la valeur de ces deux propriétés, vous pouvez utiliser l'énumération `ConnectionState`.

L'événement `InfoMessage` est déclenché lorsque le serveur vous informe d'une situation, anormale, mais qui ne justifie pas le déclenchement d'une exception (sévérité du message inférieure à 10). Le gestionnaire d'événements associé reçoit un paramètre de type `InfoMessageEventArgs`. Par la propriété `Errors` de ce paramètre, vous avez accès à des objets `SqlErrors` correspondant aux informations envoyées par le serveur. Le code suivant affiche, sur la console, les messages d'informations en provenance du serveur.

```
Private Sub ctn_InfoMessage(ByVal sender As Object, ByVal e As System.Data.
SqlClient.SqlInfoMessageEventArgs) Handles ctn.InfoMessage
        Dim info As SqlClient.SqlError
        For Each info In e.Errors
            Console.WriteLine(info.Message)
        Next
    End Sub
```

2. Exécution d'une commande

Après avoir établi une connexion vers un serveur de base de données, vous pouvez lui transmettre des instructions SQL. La classe`SqlCommand` est utilisée pour demander au serveur l'exécution d'une commande SQL. Cette classe contient plusieurs méthodes permettant l'exécution de différents types de requêtes SQL. La classe `SqlCommand` peut être instanciée de façon classique, en utilisant un de ses constructeurs où une instance peut être obtenue par la méthode `CreateCommand` de la connexion.

a. Création d'une commande

La première possibilité pour créer une `SqlCommand` est d'utiliser un des constructeurs de la classe. L'utilisation du constructeur par défaut vous oblige par la suite à utiliser différentes propriétés, pour fournir les informations concernant l'instruction SQL à exécuter.

La propriété`CommandText` contient le texte de l'instruction SQL à exécuter. La propriété `Connection` doit faire référence à une connexion valide vers le serveur de base de données. Le code suivant résume ces différentes opérations :

```
Dim cmd As SqlCommand
cmd = New SqlCommand
cmd.Connection = ctn
cmd.CommandText = " SELECT * FROM Products"
```

La deuxième solution est d'utiliser un constructeur surchargé, acceptant comme paramètres, l'instruction SQL sous forme d'une chaîne de caractères et la connexion utilisée par cette SqlCommand. Le code précédent peut donc se résumer à la ligne suivante :

```
Dim cmd As new SqlCommand( " SELECT * FROM Products",ctn)
```

La troisième solution est d'utiliser la méthode CreateCommand de la connexion. Dans ce cas, seule l'instruction SQL a besoin d'être spécifiée par la suite.

```
Dim cmd as SqlCommand
cmd = ctn.CreateCommand
cmd.CommandText = " SELECT * FROM Products"
```

b. Lecture d'informations

Fréquemment, l'instruction SQL d'une SqlCommand sélectionne un ensemble d'enregistrements dans la base, ou éventuellement une valeur unique étant le résultat d'un calcul effectué sur des valeurs contenues dans la base. Une instruction SQL, renvoyant un ensemble d'enregistrements, doit être exécutée par la méthodeExecuteReader. Cette méthode retourne un objet DataReader qui va permettre, par la suite, la lecture des informations en provenance de la base de données. Si l'instruction SQL ne renvoie qu'une valeur unique, la méthodeExecuteScalar se charge de l'exécution et retourne elle-même la valeur en provenance de la base de données.

Le code suivant permet la récupération du nombre de commandes passées par un client :

```
cmd.CommandText = " select count(orderid) from orders where customerid='FRANK'"
Console.WriteLine("le client FRANK a passé {0} commande(s)", cmd.ExecuteScalar())
```

Le cas d'instructions renvoyant plusieurs enregistrements est un peu plus complexe. Après avoir exécuté l'instruction par la méthode ExecuteReader et récupéré l'objet DataReader vous pouvez utiliser ce dernier pour parcourir les résultats renvoyés. La méthode Read de la classe DataReader permet le déplacement dans l'ensemble des enregistrements renvoyés. Cette méthode retourne un boolean indiquant s'il reste un enregistrement suivant. Le déplacement n'est possible que du premier au dernier enregistrement. Ce type de déplacement est appelé Forward Only. Les informations contenues dans l'enregistrement courant sont accessibles par une des méthodes Get... de la classeDataReader. Ces méthodes permettent d'extraire les données de l'enregistrement et de les convertir dans un type de données .NET. Il en existe une version pour chaque type de données du Framework .NET. Il faut bien sûr que les informations présentes dans l'enregistrement, puissent être converties dans le type correspondant. Si la conversion est impossible, il y a déclenchement d'une exception. Les méthodes Get... attendent, comme paramètre, le numéro du champ à partir duquel elles récupèrent l'information. Vous pouvez aussi utiliser la propriété, par défaut, Item du DataReader en indiquant le nom du champ concerné. Il n'y a pas, dans ce cas, de conversion et la valeur renvoyée est de type Object.

Le code suivant affiche la liste de toutes les catégories de produits disponibles :

```
Imports System.Data.SqlClient
Module TestExecuteReader
    Dim cmd As SqlCommand
    Dim  ctn As SqlConnection
    Dim lecteur As SqlDataReader

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " select * from categories"
        lecteur = cmd.ExecuteReader
        Do While lecteur.Read
            Console.WriteLine("numero de la categorie:{0}" & vbTab &
"Description:{1}", lecteur.GetInt32(0), lecteur("CategoryName"))
        Loop
        lecteur.Close()
        ctn.Close()
    End Sub
End Module
```

L'utilisation d'une connexion par un DataReader s'effectue de manière exclusive. Pour que la connexion soit à nouveau disponible pour une autre commande, vous devez obligatoirement fermer le DataReader après son utilisation.

c. Modification des informations

La modification des informations dans une base de données s'effectue principalement par les instructions SQL INSERT, UPDATE, DELETE. Ces instructions ne retournent pas d'enregistrements en provenance de la base de données. Pour utiliser ces instructions, vous devez créer une SqlCommand, puis demander l'exécution de cette commande par la méthodeExecuteNonQuery. Cette méthode retourne le nombre d'enregistrements affectés par l'exécution de l'instruction SQL contenue dans la SqlCommand. Si la propriété CommandText contient plusieurs instructions SQL, alors la valeur renvoyée par la méthode ExecuteNonQuery correspond au nombre total de lignes affectées par toutes les instructions SQL de la SqlCommand.

Le code suivant ajoute une nouvelle entreprise de livraison dans la table **Shippers** :

```
Imports System.Data.SqlClient
Module TestExecuteNonQuery
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Public Sub main()
        ctn = New SqlClient.SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true"
        ctn.Open()
        cmd = New SqlClient.SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = "Insert into shippers (companyname,phone) values
('DHL','02 40 41 42 43')"
        Console.WriteLine("{0} ligne(s) ajoutée(s) dans la table",
cmd.ExecuteNonQuery)
        ctn.Close()
    End Sub
End Module
```

d. Utilisation de paramètres

La manipulation d'instructions SQL peut être facilitée par la création de paramètres. Ils permettent de construire des instructions SQL génériques, pouvant facilement être réutilisées. Le principe de fonctionnement est semblable aux procédures et fonctions de Visual Basic. Une alternative à l'utilisation de paramètres pourrait être la construction dynamique d'instruction SQL par concaténation de chaînes de caractères.

Ci-dessous, un exemple utilisant cette technique et permettant la recherche d'un client par son code (nous verrons ensuite comment améliorer ce code en utilisant des paramètres) :

```
Imports System.Data.SqlClient
Module TestRequeteConcat
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim lecteur As SqlDataReader
    Dim codeClient As String
    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        Console.Write("saisir le code du client recherche :")
        codeClient = Console.ReadLine()
        cmd.CommandText = " SELECT * from Customers WHERE CustomerID = '"
& codeClient & "'"
        lecteur = cmd.ExecuteReader
        Do While lecteur.Read
            Console.WriteLine("nom du client:{0}", lecteur("ContactName"))
        Loop
        lecteur.Close()
        ctn.Close()
```

```
        Console.ReadLine()
    End Sub
End Module
```

La partie importante de ce code se situe lors de l'affectation d'une valeur à la propriété `CommandText`. Une instruction SQL correcte doit être construite par concaténation de chaînes de caractères. Dans notre cas, c'est relativement simple puisqu'il n'y a qu'une valeur variable dans l'instruction SQL, mais si plusieurs informations doivent varier, il y a une multitude de concaténations à réaliser. Les erreurs classiques dans ces concaténations sont :

- l'oubli d'un espace ;
- l'oubli des caractères ' ' pour encadrer une valeur de type chaîne de caractères ;
- un nombre de caractère ' impair.

Toutes ces erreurs ont, pour même effet, la création d'une instruction SQL invalide qui sera rejetée à l'exécution par le serveur.

L'utilisation des paramètres simplifie considérablement l'écriture de ce type de requête. Les paramètres sont utilisés pour marquer un emplacement dans une requête où sera placé, au moment de l'exécution, une valeur littérale chaîne de caractères ou numérique. Les paramètres peuvent être nommés ou anonymes. Un paramètre anonyme est introduit dans une requête par le caractère ?. Les paramètres nommés sont spécifiés par le caractère @ suivi du nom du paramètre.

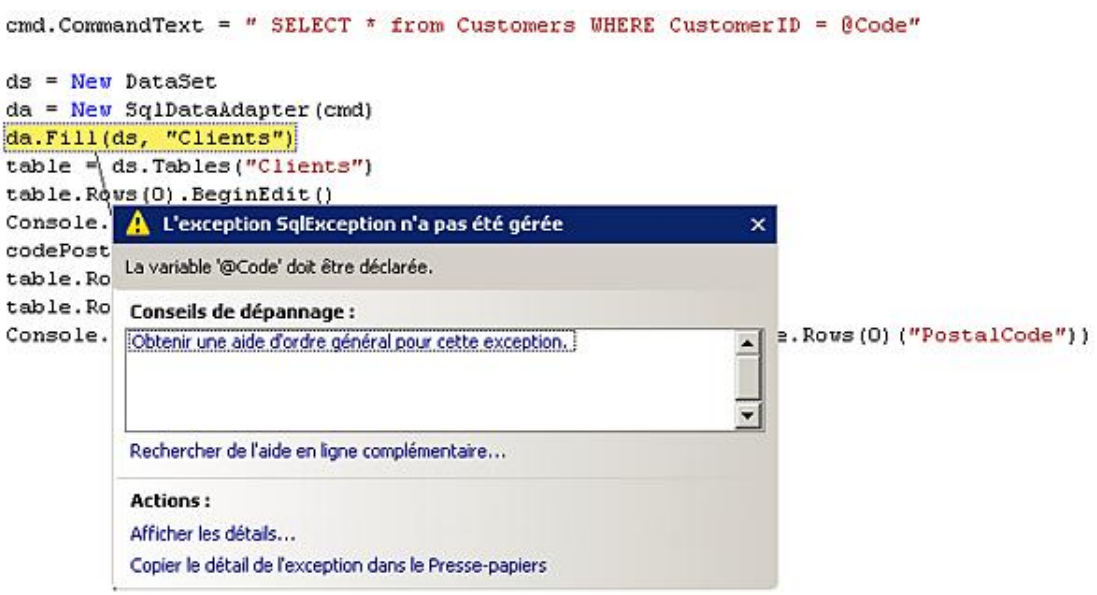
La requête de notre exemple précédent peut prendre les formes suivantes :

```
cmd.CommandText = " SELECT * from Customers WHERE CustomerID = ?"
```

ou

```
cmd.CommandText = " SELECT * from Customers WHERE CustomerID = @Code”
```

L'exécution de la `SqlCommand` échoue maintenant si aucune information n'est fournie pour le ou les paramètres.



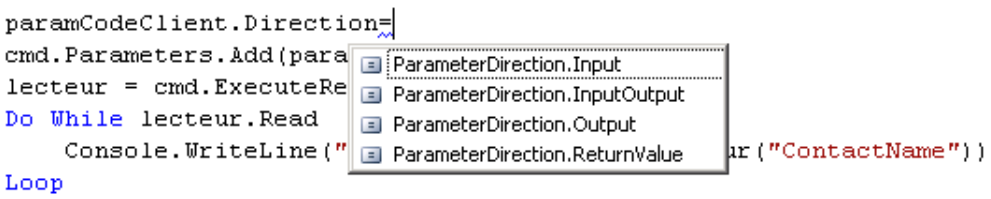
La `SqlCommand` doit avoir une liste de valeurs utilisées pour le remplacement des paramètres, au moment de l'exécution. Cette liste est stockée dans la collection **Parameters** de la `SqlCommand`. Avant l'exécution de la `SqlCommand`, il faut donc créer les objets `SqlParameter` et les ajouter à la collection. Pour chaque`SqlParameter`, il faut fournir :

- le nom du paramètre ;
- la valeur du paramètre ;
- la direction d'utilisation du paramètre.

Les deux premières informations sont indiquées lors de la construction de l'objet :

```
Dim paramCodeClient As SqlParameter
paramCodeClient = New SqlParameter("@Code", codeClient)
```

La direction d'utilisation indique si l'information contenue dans le paramètre, est passée au code SQL pour son exécution (`Input`) ou si c'est l'exécution du code SQL qui va modifier la valeur du paramètre (`Output`) ou les deux (`InputOutput`). La propriété `Direction` de la classe `SqlParameter` indique le mode d'utilisation du paramètre.



Le paramètre est maintenant prêt à être ajouté à la collection **Parameters**. Il convient d'être vigilant à ce niveau, si la requête utilise les paramètres anonymes. Les paramètres doivent obligatoirement être ajoutés à la collection, dans l'ordre de leur apparition dans la requête. Si les paramètres nommés sont utilisés, il n'est pas indispensable de respecter cette règle, mais il est prudent de s'y conformer, si un jour le code SQL est modifié et n'utilise plus les paramètres nommés. Ceci pourra être le cas si vous devez changer de type fournisseur de données et que le nouveau n'accepte pas les paramètres nommés dans une instruction SQL. La `SqlCommand` est maintenant prête pour l'exécution. À noter qu'avec cette solution nous n'avons pas à nous soucier du type de valeur attendue par l'instruction SQL pour savoir si nous devons l'encadrer avec des caractères `'`. Si des paramètres sont utilisés en sortie de l'instruction SQL, ils ne seront disponibles qu'après la fermeture du `DataReader`. L'exemple suivant affiche en plus du nom du client, le nombre de commandes qu'il a déjà passées :

```
Imports System.Data.SqlClient
Module TestRequeteConcat
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim lecteur As SqlDataReader
    Dim codeClient As String
    Dim paramCodeClient As SqlParameter
    Dim paramNbCommandes As SqlParameter

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        Console.Write("saisir le code du client recherche :")
        codeClient = Console.ReadLine()
        cmd.CommandText = " SELECT * from Customers WHERE CustomerID =
@Code;select @nbCmd=count(orderid) from orders where customerid=@code"
        paramCodeClient = New SqlParameter("@Code", codeClient)
        paramCodeClient.Direction = ParameterDirection.Input
        cmd.Parameters.Add(paramCodeClient)
        paramNbCommandes = New SqlParameter("@nbCmd", Nothing)
        paramNbCommandes.Direction = ParameterDirection.Output
        cmd.Parameters.Add(paramNbCommandes)
        lecteur = cmd.ExecuteReader
        Do While lecteur.Read
            Console.WriteLine("nom du client:{0}”, lecteur("ContactName"))
        Loop
        lecteur.Close()
        Console.WriteLine("ce client a passe {0} commande(s)", cmd.Parameters
("@nbCmd").Value)
        ctn.Close()
        Console.ReadLine()
    End Sub
End Module
```

e. Exécution de procédure stockée

Les procédures stockées sont des éléments d'une base de données correspondant à un ensemble d'instructions SQL, pouvant être exécutées par simple appel de leur nom. Ce sont des véritables programmes SQL pouvant recevoir des paramètres et renvoyer des valeurs. De plus, les procédures stockées sont enregistrées dans le cache mémoire du serveur, sous forme compilée lors de leur première exécution, ce qui accroît les performances pour les exécutions suivantes. Un autre avantage des procédures stockées est de centraliser sur le serveur de base de données tous les codes SQL d'une application. Si des modifications doivent être apportées dans les instructions SQL, vous n'aurez que des modifications à effectuer sur le serveur sans avoir à reprendre le code de l'application, donc sans avoir à regénérer et redéployer l'application.

L'appel à une procédure stockée, à partir de Visual Basic, est pratiquement similaire à l'exécution d'une instruction SQL. La propriété `CommandText` contient le nom de la procédure stockée. Vous devez également modifier la propriété `CommandType` avec la valeur `CommandType.StoredProcedure` pour indiquer que la propriété `CommandText` contient le nom d'une procédure stockée. Comme pour une instruction SQL, une procédure stockée peut utiliser des paramètres en entrée ou en sortie. Il y a un troisième type de paramètre disponible pour les procédures stockées le type `ReturnValue`. Ce type de paramètre sert à récupérer la valeur renvoyée par l'instruction `Return` de la procédure stockée (même principe qu'une fonction Visual Basic). Pour tester ces nouvelles notions, nous allons utiliser la procédure stockée suivante, qui retourne le montant total de toutes les commandes passées par un client.

```
CREATE PROCEDURE TotalClient  @code nchar(5) AS
declare @total money
select @total=sum(UnitPrice*Quantity*(1-Discount)) from Orders,[Order Details]
where customerid=@code and Orders.orderid=[order details].orderid
return @total
GO
```

Au niveau du code Visual Basic, nous devons indiquer qu'il s'agit de l'exécution d'une procédure stockée et ajouter un paramètre pour récupérer la valeur de retour de la procédure stockée. Ce paramètre doit s'appeler `RETURN_VALUE`.

```
Imports System.Data.SqlClient
Module TestProcedureStockee
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim paramCodeClient As SqlParameter
    Dim paramMontant As SqlParameter
    Dim codeclient As String

    Public Sub main()
        Console.Write("saisir le code du client recherche :")
        codeClient = Console.ReadLine()
        ctn = New SqlConnection()
        ctn.ConnectionString = “Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true”
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = "TotalClient"
        cmd.CommandType = CommandType.StoredProcedure
        paramCodeClient = New SqlParameter("@Code", codeClient)
        paramCodeClient.Direction = ParameterDirection.Input
        cmd.Parameters.Add(paramCodeClient)
        paramMontant = New SqlParameter("RETURN_VALUE", SqlDbType.Decimal)
        paramMontant.Direction = ParameterDirection.ReturnValue
        cmd.Parameters.Add(paramMontant)
        cmd.ExecuteNonQuery()
        Console.WriteLine("Ce client a passe pour {0} Euros de commande",
paramMontant.Value)
        Console.ReadLine()
        ctn.Close()
    End Sub
End Module
```

Utilisation du mode non connecté

Dans un mode non connecté, la liaison avec le serveur de base de données n'est pas permanente. Il faut donc conserver localement les données sur lesquelles on souhaite travailler. L'idée est de recréer, à l'aide de différentes classes, une organisation similaire à celle d'une base de données. Les principales classes sont représentées sur le schéma suivant :

`DataSet`

C'est le conteneur de plus haut niveau, il joue le même rôle que la base de données.

`DataTable`

Comme son nom l'indique, c'est l'équivalent d'une table de base de données.

`DataRow`

Cette classe joue le rôle d'un enregistrement (ligne).

`DataColumn`

Cette classe remplace un champ (colonne) d'une table.

`UniqueConstraint`

C'est l'équivalent de la clé primaire d'une table.

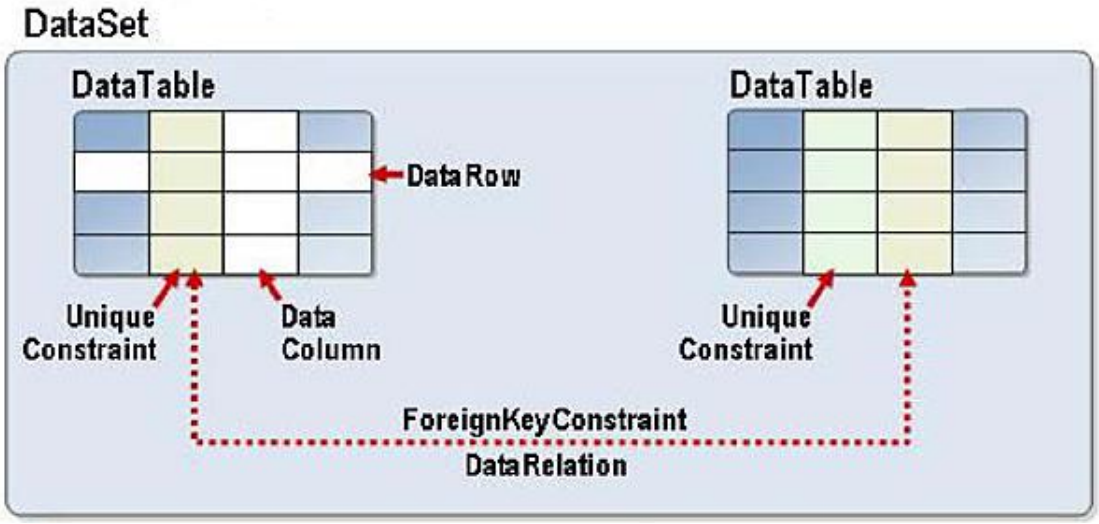
`ForeignKeyConstraint`

C'est l'équivalent de la clé étrangère.

`DataRelation`

Représente un lien parent/enfant entre deux DataTable.

Le schéma ci-dessous représente cette organisation.



Nous allons voir maintenant comment créer et manipuler toutes ces classes.

1. Remplir un DataSet à partir d'une base de données

Pour pouvoir travailler localement avec les données, nous devons les rapatrier depuis la base de données dans un `DataSet`. Chaque fournisseur de données fournit une classe `DataAdapter`, assurant le dialogue entre la base de données et un `DataSet`. Tous les échanges se font par l'intermédiaire de cette classe, aussi bien de la base vers le

DataSet que du DataSet vers la base pour la mise à jour des données. Le DataAdapter utilisera une connexion pour contacter le serveur et une ou plusieurs commandes pour le traitement des données.

a. Utilisation d'un DataAdapter

La première chose à réaliser est de créer une instance de la classe SQLDataAdapter. Nous devons ensuite configurer le DataAdapter afin de lui indiquer quelles données nous souhaitons rapatrier à partir de la base de données. La propriété SelectCommand doit référencer un objet Command, contenant l'instruction SQL chargée de sélectionner les données. L'objet Command utilisé peut également appeler une procédure stockée. La seule contrainte est que l'instruction SQL exécutée par l'objet Command soit une instruction SELECT. La classe DataAdapter contient également les propriétés InsertCommand, DeleteCommand et UpdateCommand référençant les objets Command, utilisés lors de la mise à jour de la base de données. Tant que nous ne souhaitons pas effectuer de mise à jour de la base, ces propriétés sont facultatives. Elles seront étudiées plus en détail dans le chapitre consacré à la mise à jour de la base de données.

La méthode Fill de la classe DataAdapter est ensuite utilisée pour remplir le DataSet avec le résultat de l'exécution de la commande SelectCommand. Cette méthode attend, comme paramètre, le DataSet qu'elle doit remplir et un objet DataTable ou une chaîne de caractères utilisée pour nommer la DataTable dans le DataSet. Le DataAdapter utilise, en interne, un objet DataReader pour obtenir le nom des champs et le type des champs pour créer la DataTable dans le Dataset et ensuite le remplir avec les données. La DataTable et les DataColumn sont créés uniquement s'ils n'existent pas déjà, sinon la méthode Fill utilise la structure existante. Si une DataTable est créée, elle est ajoutée à la collection**Tables** du DataSet. Le type de données des DataColumn est défini en fonction des mappages prévus par le fournisseur de données, entre les types de la base de données et les types .NET. L'exemple suivant remplit un DataSet avec les code, nom, adresse et ville des clients.

```
Imports System.Data.SqlClient
Module TestDataSet1
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true”
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT CustomerId,ContactName,Address,city from
Customers"
        ds = New DataSet
        da = New SqlDataAdapter()
        da.SelectCommand = cmd
        da.Fill(ds, "Customers")
    End Sub
End Module
```

Dans ce code, la connexion n'a pas été ouverte et fermée, explicitement. En effet, la méthode Fill ouvre la connexion si elle n'est pas déjà ouverte et dans ce cas, la referme également à la fin de son exécution. Toutefois, si vous avez besoin d'utiliser plusieurs fois la méthode Fill, il est plus efficace de gérer vous-même l'ouverture et la fermeture de connexion. Dans tous les cas, la méthode Fill laisse la connexion dans l'état où elle l'a trouvée.

Un DataSet peut bien sûr contenir plusieurs DataTable créées à partir de DataAdapter différents. Les données peuvent même provenir de bases de données différentes, voire de types de serveurs différents.

Lorsque le DataAdapter construit la DataTable, les noms des champs de la base sont utilisés pour nommer les DataColumn. Il est possible de personnaliser ces noms en créant des objets DataTableMapping et en les ajoutant à la collection **TableMappings** du DataAdapter. Ces objets DataTableMapping contiennent eux-mêmes des objets DataColumnMapping utilisés par la méthode Fill, comme traducteurs entre les noms des champs dans la base et les noms des Datacolumn dans le DataSet. Dans ce cas, lors de l'appel de la méthode Fill nous devons lui indiquer le nom du DataTableMapping à utiliser.

Si, pour un ou plusieurs champs, il n'y a pas de mappage disponible, alors le nom du champ dans la base est utilisé comme nom pour la DataColumn correspondante. Nous pouvons, par exemple, utiliser cette technique pour traduire les champs de la base **Northwind**.

Le code suivant effectue cette traduction et affiche le nom des Datacolumn du DataTable créé :

```
Imports System.Data.SqlClient
Imports System.Data.Common
```

```
Module TestTableMapping
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim mappage As DataTableMapping
    Dim dc As DataColumn

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT CustomerId,ContactName,Address,city from
Customers”
        ds = New DataSet
        da = New SqlDataAdapter()
        da.SelectCommand = cmd
        mappage = New DataTableMapping("Customers", "Clients")
        mappage.ColumnMappings.Add("CustomerId", "CodeClient")
        mappage.ColumnMappings.Add("ContactName", "Nom")
        mappage.ColumnMappings.Add("Address", "Adresse")
        mappage.ColumnMappings.Add("city", "Ville")
        da.TableMappings.Add(mappage)
        da.Fill(ds, "Customers")
        For Each dc In ds.Tables("Clients").Columns
           Console.Write(dc.ColumnName & vbTab)
        Next
        Console.ReadLine()
    End Sub
End Module
```

Nous obtenons à l’affichage :

```
CodeClient      Nom     Adresse Ville
```

b. Ajout de contraintes existantes à un DataSet

La méthode `Fill` ne fait que transférer, vers le `DataSet`, les données en provenance de la base. Bien souvent, des contraintes de clés primaires sont utilisées dans la base de données et, par défaut, la méthode `Fill` ne les rapatrie pas dans le `DataSet`. Pour pouvoir récupérer ces contraintes dans le `DataSet`, il y a deux solutions possibles :

- Modifier la propriété `MissingSchemaAction` du `DataAdapter` avec la valeur `MissingSchemaAction.AddWithKey`

```
da.MissingSchemaAction = MissingSchemaAction.AddWithKey
```

- Procéder en deux étapes en appelant d’abord la méthode`FillSchema` du `DataAdapter` pour créer la structure complète de la `DataTable`, puis ensuite appeler la méthode `Fill` pour remplir la `DataTable` avec les données.

```
da.FillSchema(ds, SchemaType.Mapped, "Customers")
da.Fill(ds, "Customers")
```

Le deuxième paramètre de la méthode `FillSchema` indique si le mappage doit être pris en compte ou si les informations issues de la base sont utilisées.

Il est important d’ajouter les contraintes de clés primaires car la méthode `Fill` va se comporter différemment si elles existent ou pas.

Si les contraintes existent au niveau du `DataSet`, lorsque la méthode `Fill` importe un enregistrement depuis la base, elle vérifie s’il n’existe pas déjà une ligne avec la même valeur de clé primaire dans la `DataTable`. Si c’est le cas, elle met uniquement à jour les champs de la ligne existante. Si, par contre, il n’y a pas de ligne avec une valeur de clé primaire identique, alors la ligne est créée dans la `DataTable`.

S’il n’y a pas de contrainte de clé primaire sur la `DataTable`, la méthode `Fill` ajoute tous les enregistrements en provenance de la base. Il risque dans ce cas d’y avoir des doublons dans la `DataTable`. Ceci est particulièrement

important lorsque la méthode Fill doit être appelée plusieurs fois pour, par exemple, obtenir les données modifiées par une autre personne dans la base de données.

2. Configurer un DataSet sans base de données

Il n'est pas nécessaire de disposer d'une base de données pour pouvoir utiliser des DataSet. Ils peuvent servir d'alternative à l'utilisation de tableaux pour la gestion interne des données d'une application. Dans ce cas, toutes les opérations effectuées automatiquement par le DataAdapter devront être réalisées manuellement par le code. Ceci inclut notamment la création des DataTable avec leurs DataColumn. La première opération à réaliser est de créer une instance de la classe DataTable. Le constructeur attend, comme paramètre, le nom de la DataTable. Ce nom est ensuite utilisé pour identifier la DataTable dans la collection Tables du DataSet. Après sa création, la DataTable ne contient aucune structure. Nous devons donc créer une ou plusieurs DataColumn et les ajouter à la collection **Columns** de la DataTable.

Les DataColumn peuvent être créées, en utilisant un des constructeurs de la classe, ou automatiquement lors de l'ajout à la collection **Columns**. La première solution fournit plus de souplesse puisqu'elle permet la configuration de nombreuses propriétés de la DataColumn au moment de sa création. Vous devez au minimum indiquer un nom et un type de données pour la DataColumn.

```
col = New DataColumn(("Ht", Type.GetType("int"))
table.Columns.Add(col)
table.Columns.Add("Tva", Type.GetType("decimal"))
```

Une DataColumn peut également être construite comme étant une expression basée sur une ou plusieurs autres DataColumn. Vous devez, dans ce cas, indiquer lors de la création de la DataColumn, l'expression servant au calcul de sa valeur. Le type de données généré par l'expression doit bien sûr être compatible avec le type de données de la DataColumn. Vous devez également être vigilant dans la conception de l'expression, en respectant la casse et en veillant à ne pas créer de référence circulaire entre les DataColumn.

```
table.Columns.Add("Ttc", Type.GetType("System.Decimal"), "Ht * (1 + (Tva /100))")
```

Pour assurer l'unicité des valeurs d'une DataColumn, il est possible d'utiliser un type de DataColumn auto incrémenté. La propriété AutoIncrement de cette DataColumn doit être positionnée sur **true**. Vous pouvez également modifier le pas d'incrémentation avec la propriété AutoIncrementStep et la valeur de départ avec la propriété AutoIncrementSeed. La valeur contenue dans cette DataColumn est calculée automatiquement lors de l'ajout d'une ligne à une DataTable en fonction de ces propriétés et des lignes existant déjà dans la DataTable.

Ce type de DataColumn est généralement utilisé comme clé primaire d'une DataTable. Vous avez la possibilité de définir la clé primaire d'une DataTable en fournissant à la propriété PrimaryKey un tableau contenant les différentes DataColumn devant composer la clé primaire. Les DataColumn concernées verront certaines de leurs propriétés automatiquement modifiées. La propriété Unique sera positionnée sur **true** et la propriété AllowDBNull sur **false**. Si la clé primaire est constituée de plusieurs DataColumn, seule la propriété AllowDBNull sera modifiée sur ces DataColumn.

```
col = New DataColumn("Numero", Type.GetType("System.Int32"))
col.AutoIncrement = True
col.AutoIncrementSeed = 1000
col.AutoIncrementStep = 1
table.Columns.Add(col)
table.PrimaryKey = New DataColumn() {col}
```

3. Manipuler les données dans un DataSet

Quelle que soit la méthode utilisée pour remplir un DataSet, le but de toute application est de manipuler les données présentes dans le DataSet. La classe DataTable contient de nombreuses propriétés et méthodes facilitant la manipulation des données.

a. Lecture des données

La lecture des données est l'opération la plus fréquente réalisée sur un DataSet. Il faut tout d'abord obtenir une référence sur la DataTable contenant les données, puis nous pouvons parcourir la collection **Rows** de la DataTable. Cette collection est une instance de la classe DataRowCollection. Elle dispose de la propriété Item, par défaut, permettant l'accès à une ligne particulière par un index. La propriété count permet de connaître le nombre de lignes disponibles. Les habitués de ADO seront un peu perdus au début, car dans une DataTable, il n'y a pas de notion de pointeur d'enregistrement, d'enregistrement courant, de méthodes de déplacement dans le jeu de résultats. Si vous

voulez gérer toutes ces notions, vous devez le prévoir explicitement dans votre code. La méthode GetEnumerator met à notre disposition une instance de classe implémentant l'interface IEnumerator. Par cette instance de classe, nous avons accès aux méthodes MoveNext et Reset ainsi qu'a la propriété Current. Ces trois éléments permettent de parcourir facilement toutes les lignes de la DataTable. Chaque ligne correspond à une instance de la classe DataRow. Cette classe possède également une propriété Item, par défaut, fournissant un accès aux différents champs de la DataRow. Chaque champ peut être obtenu par son nom ou par son index.

Le code suivant illustre ces notions en affichant la liste des clients :

```
Imports System.Data.SqlClient
Module TestLectureDataTable

    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim en As IEnumerator

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT ContactTitle,ContactName from Customers"
        ds = New DataSet
        da = New SqlDataAdapter()
        da.SelectCommand = cmd
        da.Fill(ds, "Customers")
        ' on recupere l'enumerateur sur les lignes de la DataTable
        en = ds.Tables("Customers").Rows.GetEnumerator
        ' on se replace au debut de la table (par securite)
        en.Reset()
        ' on boucle tant que la méthode MoveNext nous indique qu'il reste des
lignes
        Do While en.MoveNext
            ' on accede aux champs par le nom
            Console.Write(en.Current("ContactName") & vbTab)
            ' ou par le numero
            Console.WriteLine(en.Current(0))
        Loop
        Console.ReadLine()
    End Sub
End Module
```

b. Création de contraintes sur une DataTable

Vous pouvez utiliser des contraintes pour mettre en œuvre des restrictions sur les données présentes dans une DataTable. Les contraintes constituent des règles qui sont appliquées à une DataColumn ou à ses DataColumn liées. Elles déterminent les actions effectuées lorsque la valeur contenue dans une ligne est modifiée. Elles sont prises en compte pour un DataSet, uniquement, si sa propriété EnforceConstraints est positionnée sur **true**.

Deux types de contraintes sont utilisables :

UniqueConstraint

Ce type de contrainte va garantir que la ou les valeurs présentes dans une DataColumn ou un groupe de DataColumn sont uniques. La mise en place d'une contrainte unique s'effectue en créant une instance de la classe UniqueConstraint avec la liste des DataColumn concernées par la contrainte. Cette UniqueConstraint doit, ensuite, être ajoutée à la collection **Constraints** de la DataTable.

```
table.Constraints.Add(New UniqueConstraint(New DataColumn() {col}))
```

Si la contrainte ne porte que sur une DataColumn, il est aussi possible de modifier simplement la propriété Unique de cette DataColumn sur **true**, pour créer une contrainte unique. À noter également que la création d'une clé primaire génère automatiquement une contrainte unique, par l'inverse n'est pas vrai. La violation de la contrainte, à la suite de la modification d'une ligne, déclenche une exception.

ForeignKeyConstraint

Les ForeignKeyConstraint contrôlent comment vont se comporter les DataTable liées lors de la modification ou de la suppression d'une valeur dans la DataTable principale. Une action différente peut être envisagée pour une suppression et une modification. La classe ForeignKeyConstraint dispose des propriétés DeleteRule et UpdateRule indiquant le comportement, lors de la suppression ou de la modification. Les valeurs suivantes sont possibles :

Cascade

La suppression ou modification est propagée à la ou aux lignes liées.

SetNull

La valeur est modifiée à DBNull dans les lignes liées.

SetDefault

La valeur par défaut est prise dans les lignes liées.

None

Aucune action n'est effectuée sur les lignes liées.

L'ajout d'une ForeignkeyConstraint se fait par la création d'une instance de la classe en lui indiquant la ou les DataColumn de DataTable parent et la ou les DataColumn de la table enfant. Si plusieurs DataColumn font partie de la contrainte, elles sont fournies sous forme de tableau.

Le code suivant ajoute une contrainte entre la DataTable **Factures** et la DataTable **LignesFacture**, pour que la suppression d'une facture entraîne la suppresion de toutes ses lignes.

```
fkFact_LignesFact = New ForeignKeyConstraint("FK_FACT_LIGNESFACT",
ds.Tables("Factures").Columns("Numero"),
ds.Tables("LignesFacture").Columns("NumFact"))
fkFact_LignesFact.AcceptRejectRule = AcceptRejectRule.Cascade
  fkFact_LignesFact.DeleteRule = Rule.Cascade
 ds.EnforceConstraints = True
```

c. Ajout de relations entre les DataTables

Dans un DataSet contenant plusieurs DataTable, vous pouvez ajouter des relations entre les DataTable. Ces relations permettent la navigation entre les lignes des différentes DataTable. Une instance de la classeDataRelation doit être créée et ajoutée à la collection Relations du DataSet. La création peut se faire directement par la méthode Add de la collection DataRelations. Les informations à fournir sont :

- Le nom de la relation permettant de retrouver par la suite la DataRelation dans la collection.
- La ou les DataColumn parentes sous forme d'un tableau de DataColumn s'il y en a plusieurs.
- La ou les DataColumn enfants sous forme d'un tableau, s'il y en a plusieurs.

Le code suivant ajoute une relation entre la table **Customers** et la table **Orders** :

```
ds.Relations.Add("Client_Commandes", ds.Tables("Customers").
Columns("CustomerId"),
ds.Tables("Orders").Columns("CustomerId"))
```

À noter que les DataRelation fonctionnent parallèlement avec les ForeignKeyConstaint et les UniqueConstraint. Par défaut, la création de la relation va placer une UniqueConstraint sur la table parent et une ForeignKeyConstraint sur la table enfant. Si vous ne souhaitez pas que ces contraintes soient ajoutées automatiquement si elles n'existent pas, vous devez ajouter un boolean **false** comme quatrième paramètre, lors de l'ajout de la DataRelation.

d. Parcourir les relations

Le but principal des relations est de permettre la navigation d'une DataTable vers une autre à l'intérieur d'un DataSet. Nous pouvons ainsi obtenir tous les objets DataRow d'une table liés à une DataRow d'une autre DataTable. Par exemple, après avoir chargé les tables **Customers** et **Orders** dans le DataSet et établi une relation entre ces deux tables, nous pouvons, à partir d'une ligne de la DataTable **Clients** obtenir depuis la DataTable **Orders** toutes les commandes de ce client. La méthode GetChildRows retourne, sous forme d'un tableau de DataRow, toutes les lignes contenant les commandes de ce client.

Cette méthode prend, comme paramètre, le nom de la DataRelation utilisée pour le lien. L'exemple du code suivant met cela en application, en affichant pour chaque client le numéro et la date de ses commandes :

```
Imports System.Data.SqlClient
Module TestRelations
    Dim cmdCustomers, cmdOrders As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim daCustomers, daOrders As SqlDataAdapter
    Dim ligneClient, ligneCommandes As DataRow

    Public Sub main()
        ds = New DataSet
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        cmdCustomers = New SqlCommand
        cmdCustomers.Connection = ctn
        cmdCustomers.CommandText = " SELECT * from Customers”
        daCustomers = New SqlDataAdapter()
        daCustomers.SelectCommand = cmdCustomers
        daCustomers.Fill(ds, "Customers")
        cmdOrders = New SqlCommand
        cmdOrders.Connection = ctn
        cmdOrders.CommandText = " SELECT * from Orders"
        daOrders = New SqlDataAdapter()
        daOrders.SelectCommand = cmdOrders
        daOrders.Fill(ds, "Orders")
        ds.Relations.Add("Client_Commandes", ds.Tables("Customers").Columns
("CustomerId"), ds.Tables("Orders").Columns("CustomerId"))
        For Each ligneClient In ds.Tables("Customers").Rows
          Console.WriteLine(ligneClient("ContactName"))
          For Each ligneCommandes In ligneClient.GetChildRows("Client_Commandes")
               Console.WriteLine(vbTab & "commande N {0} du {1}",
ligneCommandes("OrderId"), ligneCommandes("OrderDate"))
            Next
        Next
    End Sub
End Module
```

La navigation d'une ligne enfant vers une ligne parent est aussi possible, en utilisant la méthode GetParentRow qui, elle aussi, attend comme paramètre le nom de la relation utilisée comme lien.

La partie du code suivant affiche pour chaque commande le nom du client l'ayant passée :

```
For Each ligneCommandes In ds.Tables("Orders").Rows
            Console.WriteLine("la commande {0} a ete passee par {1}",
ligneCommandes("OrderId"), ligneCommandes.GetParentRow("Client_Commandes")
("ContactName"))
Next
```

e. État et versions d'une DataRow

La classe DataRow est capable de suivre les différentes modifications apportées aux données qu'elle contient. La propriété RowState permet de contrôler les modifications apportées à la ligne.

Cinq valeurs définies dans une énumération sont possibles pour cette propriété :

Unchanged

La ligne n'a pas changé depuis le remplissage du DataSet par la méthode Fill ou la validation des modifications par la méthode AcceptChanges.

Added

La ligne a été ajoutée mais les modifications n'ont pas encore été validées par la méthode `AcceptChanges`.

Modified

Un ou plusieurs champs de la ligne ont été modifiés.

Deleted

La ligne a été effacée mais les modifications n'ont pas encore été validées par la méthode `AcceptChanges`.

Detached

La ligne a été créée mais ne fait pas encore partie de la collection **Rows** d'une `DataTable`.

Les différentes versions d'une ligne sont également disponibles. Lorsque vous accédez aux valeurs contenues dans une ligne, vous pouvez spécifier la version qui vous intéresse.

Pour cela, l'énumération `DataRowVersion` propose quatre valeurs :

Current

Version actuelle de la ligne. Cette version n'existe pas pour une ligne dont l'état est `Deleted`.

Default

Version par défaut de la ligne. Pour une ligne dont l'état est `Added`, `Modified`, `Unchanged`, cette version est équivalente à la version `Current`. Pour une ligne dont l'état est `Deleted`, cette version est équivalente à la version `Original`. Pour une ligne dont l'état est `Detached`, cette version est égale à la version `Proposed`.

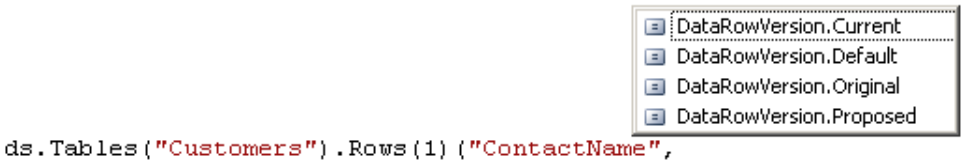
Original

Version originale de la ligne. Pour une ligne dont l'état est `Added`, cette version n'existe pas.

Proposed

Version transitoire disponible pendant une opération de modification de la ligne ou pour une ligne ne faisant pas partie d'une collection **Rows** d'une `DataTable`.

L'indication de la version désirée doit être spécifiée, lors de l'accès à un champ particulier d'une `DataRow`. Pour cela, il faut utiliser l'une des constantes précédentes, à la suite du nom ou de l'index du champ, lors de l'utilisation de la propriété `Item`, par défaut, de la `DataRow`.



Ces différentes versions seront utilisées, lors de la mise à jour de la base de données, pour par exemple gérer les accès concurrents.

f. Ajout de données

L'ajout d'une ligne à une `DataTable` s'effectue simplement en ajoutant une `DataRow` à la collection **Rows** d'une `DataTable`. Il faut, au préalable, créer une instance de la classe `DataRow`. C'est à ce niveau que nous rencontrons un problème.

Il n'y a pas de constructeur disponible pour la classe DataRow. Rassurez-vous ce n'est pas une erreur de Visual Basic, mais c'est bien volontairement qu'il n'existe pas de constructeur pour cette classe. En effet, lorsque nous avons besoin d'une nouvelle instance d'une DataRow, nous ne voulons pas une DataRow quelconque mais une DataRow spécifique au schéma de notre DataTable. C'est pour cette raison que c'est à elle qu'est confié le soin de créer l'instance dont nous avons besoin par l'intermédiaire de la méthode NewRow.

```
Dim nouvelleLigne As DataRow
nouvelleLigne = ds.Tables("Customers").NewRow()
```

L'état de cette ligne est, pour l'instant, Detached. Nous pouvons ensuite ajouter des données dans cette nouvelle ligne.

```
nouvelleLigne("ContactName") = "Dupond"
```

Après cela, il nous reste à ajouter la ligne à la collection **Rows** de la DataTable.

```
ds.Tables("Customers").Rows.Add(nouvelleLigne)
```

L'état de cette nouvelle ligne est maintenant Added.

g. Modification de données

La modification des données contenues dans une ligne est réalisée, simplement, en affectant aux champs correspondants, les valeurs souhaitées. Ces valeurs sont stockées dans la version Current de la ligne. L'état de la ligne est alors Modified. Cette solution présente un petit inconvénient. Si plusieurs champs d'une ligne doivent être modifiés, il peut y avoir pendant la modification, des états transitoires qui violent une ou plusieurs contraintes placées sur la DataTable. C'est, par exemple, le cas s'il y a sur la DataTable, une contrainte de clé primaire placée sur deux DataColumn. Ceci a pour effet de déclencher une exception. Pour pallier ce problème, nous pouvons demander temporairement l'arrêt de la vérification des contraintes pour cette ligne. La méthode BeginEdit passe la ligne en mode édition et suspend donc la vérification des contraintes pour cette ligne. Les valeurs affectées aux champs ne sont pas stockées dans la version Current de la ligne mais dans la version Proposed. Lorsque toutes les modifications sont effectuées sur la ligne, vous pouvez les valider ou les annuler en appelant la méthode EndEdit ou la méthode CancelEdit. Vous pouvez également vérifier les valeurs, en gérant l'événement ColumnChanged de la DataTable. Dans le gestionnaire d'événements, vous recevez un argument de type DataColumnChangeEventArg permettant de savoir quelle DataColumn a été modifiée (args.Column.ColumnName), la valeur proposée pour cette DataColumn (args.ProposedValue) et permettant d'annuler les modifications (args.row.CancelEdit). En cas de validation avec la méthode EndEdit, la version Proposed de la ligne est recopiée dans la version Current et l'état de la ligne devient Modified. Si, par contre, vous annulez les modifications avec la méthode CancelEdit, la version Current n'est pas modifiée et l'état de la ligne est inchangé. Dans tous les cas, après l'appel d'une de ces deux méthodes, la vérification des contraintes est réactivée.

L'exemple suivant permet la modification du code postal d'un client en vérifiant que celui-ci est bien numérique :

```
Imports System.Data.SqlClient
Module TestModificationLigne
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim codeClient As String
    Dim codePostal As String
    Dim paramCodeClient As SqlParameter
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim WithEvents table As DataTable
    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial Catalog=
Northwind;Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        Console.Write("saisir le code du client a modifier:")
        codeClient = Console.ReadLine()
        cmd.CommandText = " SELECT * from Customers WHERE CustomerID = @Code"
        paramCodeClient = New SqlParameter("@Code", codeClient)
        paramCodeClient.Direction = ParameterDirection.Input
        cmd.Parameters.Add(paramCodeClient)
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
```

```
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        table.Rows(0).BeginEdit()
        Console.Write("saisir le nouveau code postal du client:")
        codePostal = Console.ReadLine()
        table.Rows(0)("PostalCode") = codePostal
        table.Rows(0).EndEdit()
        Console.WriteLine("le nouveau code postal est : {0}",
table.Rows(0)("PostalCode"))
        Console.ReadLine()
    End Sub

    Private Sub table_ColumnChanged(ByVal sender As Object, ByVal e
As System.Data.DataColumnChangeEventArgs) Handles table.ColumnChanged
        If e.Column.ColumnName = "PostalCode" Then
            If Not IsNumeric(e.ProposedValue) Then
                e.Row.CancelEdit()
            End If
        End If
    End Sub
End Module
```

h. Suppression de données

Deux solutions différentes sont disponibles. Vous pouvez effacer une ligne ou supprimer une ligne. La nuance est subtile entre ces deux solutions :

La suppression d'une donnée se fait avec la méthode `Remove` qui retire définitivement la `DataRow` de la collection **Rows** de la `DataTable`. Cette suppression est définitive.

La méthode `Deleted` ne fait que marquer la ligne pour la supprimer ultérieurement. L'état de la ligne passe à `Deleted` et ce n'est qu'au moment de la validation des modifications que la ligne est réellement supprimée de la collection **Rows** de la `DataTable`. Si les modifications sont annulées, la ligne reste dans la collection **Rows**.

La méthode `Remove` est une méthode de la collection **Rows** (elle agit directement sur son contenu), la méthode `Delete` est une méthode de la classe `DataRow` (elle ne fait que changer une propriété de la ligne).

```
' efface la ligne
table.Rows(1).Delete()
        ' supprime la ligne
        table.Rows.Remove(table.Rows(1))
```

i. Valider ou annuler les modifications

Jusqu'à présent, les modifications effectuées sur une ligne sont temporaires, il est encore possible de revenir à la version précédente, ou au contraire de valider de façon définitive les modifications dans les lignes (mais encore dans la base). Les méthodes `AcceptChanges` ou `RejectChanges` permettent respectivement la validation ou l'annulation des modifications. Elles peuvent s'appliquer sur une `DataRow` individuelle, une `DataTable` ou un `DataSet` entier. Lorsque la méthode `AcceptChanges` est exécutée, les actions suivantes sont réalisées :

- La méthode `EndEdit` est appelée implicitement pour la ligne.
- Si l'état de la ligne était `Added` ou `Modified`, il devient `Unchanged` et la version `Current` est recopiée dans la version Originale.
- Si l'état de la ligne était `Deleted`, alors la ligne est supprimée.

La méthode `RejectChanges` exécute les actions suivantes :

- La méthode `CancelEdit` est appelée implicitement pour la ligne.
- Si l'état de la ligne était `Deleted` ou `Modified`, il revient à `Unchanged` et la version `Original` est recopiée dans la version `Current`.

- Si l'état de la ligne était Added, alors elle est supprimée.

S'il existe des contraintes de clé étrangère, l'action de la méthode AcceptChanges ou RejectChanges est propagée aux lignes enfants en fonction de la propriété AcceptRejectRule de la contrainte.

j. Filtrer et trier des données

Il est fréquent d'avoir besoin de limiter la quantité de données visibles dans une DataTable ou encore de modifier l'ordre des lignes. La première solution qui vient à l'esprit est de recréer une requête SQL avec une restriction ou une clause ORDERBY. C'est oublier que nous sommes dans un mode de fonctionnement déconnecté et qu'il est souhaitable de limiter les accès à la base, voire pire, que la base n'est pas disponible. Nous devons donc n'utiliser que les données disponibles, en faisant attention à ne pas en perdre. La classeDataView va nous être très utile pour solutionner nos problèmes. Cette classe va nous servir à modifier la vision des données dans la DataTable sans risque pour les données elles-mêmes. Il peut y avoir plusieurs DataView pour une même DataTable, elles correspondent à des points de vue différents de la DataTable. Pratiquement toutes les opérations réalisables sur une DataTable le sont aussi par l'intermédiaire d'une DataView.

Deux solutions sont disponibles pour obtenir une DataView :

- Créer une instance par l'un des constructeurs.
- Utiliser l'instance, par défaut, fournie par la propriété DefaultView.

Le premier constructeur utilisable attend simplement comme paramètre la DataTable à partir de laquelle est générée la DataView. Dans ce cas, il n'y a aucun filtre ni aucun tri d'effectué sur les données visibles par la DataView. Un résultat équivalent est obtenu en utilisant la propriétéDefaultView d'une DataTable.

Le deuxième constructeur permet de spécifier un filtre, un ordre de tri et la version des lignes concernées. Pour être visibles dans la DataView, les lignes devront correspondre à tous ces critères. Les différents critères peuvent aussi être modifiés par trois propriétés.

RowFilter

Cette propriété accepte une chaîne de caractères représentant la condition devant être remplie pour qu'une ligne soit visible. Cette condition a une syntaxe tout à fait similaire aux conditions d'une clause WHERE. Les opérateurs And et Or peuvent également être utilisés pour associer plusieurs conditions.

L'exemple suivant affiche le nom des clients commerciaux ou directeurs de vente en France :

```
Imports System.Data.SqlClient
Module TestDataView
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim table As DataTable
    Dim ligne As DataRowView
    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT * from Customers"
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        table.DefaultView.RowFilter = "Country='France' and (contactTitle=
'Sales Agent' or contactTitle='Sales Manager')"
        For Each ligne In table.DefaultView
            Console.WriteLine("nom : {0}", ligne("ContactName"))
        Next
    End Sub
End Module
```

Un filtre peut être annulé en affectant une chaîne vide à la propriété `RowFilter`.

Sort

Cette propriété accepte, elle aussi, une chaîne de caractères représentant le ou les critères utilisés pour le tri. La syntaxe est équivalente à celle de la clause `ORDER BY`.

L'exemple suivant affiche les clients triés par pays puis par nom, pour un même pays :

```
        ‘ on annule le filtre precedent
        table.DefaultView.RowFilter = ""
        ‘ toutes les lignes sont maintenant visibles
        ‘ on ajoute un critere de tri
        table.DefaultView.Sort = "Country ASC,ContactName ASC"
        For Each ligne In table.DefaultView
            Console.WriteLine("Pays : {0}" & vbTab & vbTab & " nom : {1}”, ligne("Country"),
ligne("ContactName"))
        Next
```

RowStateFilter

Cette propriété détermine l'état des lignes et quelle version de la ligne est visible dans la `DataView`. Huit possibilités sont disponibles :

CurrentRows

Présente la version `Current` de toutes les lignes ajoutées, modifiées ou inchangées.

Added

Présente la version `Current` de toutes les lignes ajoutées.

Deleted

Présente la version `Original` de toutes les lignes effacées.

ModifiedCurrent

Présente la version `Current` de toutes les lignes modifiées.

ModifiedOriginal

Présente la version `Original` de toutes les lignes modifiées.

None

Aucune ligne.

OriginalRows

Présente la version `Original` de toutes les lignes modifiées, supprimées ou inchangées.

Unchanged

Présente la version `Current` de toutes les lignes inchangées.

L'exemple suivant supprime deux lignes et les affiche par l'intermédiaire d'un filtre :

```
        ‘ on supprime deux lignes
        table.Rows(2).Delete()
        table.Rows(5).Delete()
        ‘ on annule<+>le filtre
        table.DefaultView.RowFilter = ""
        ‘ on affiche la version original des lignes supprimees
```

```
        table.DefaultView.RowStateFilter = DataViewRowState.Deleted
        For Each ligne In table.DefaultView
            Console.WriteLine("Pays : {0}" & vbTab & vbTab & " nom : {1}", ligne("Country"),
ligne("ContactName"))
        Next
```

k. Rechercher des données

La recherche peut s'effectuer avec les deux méthodes `Find` et `FindRows`. Pour que ces deux méthodes fonctionnent, il est impératif d'avoir au préalable trié les données avec la propriété `Sort`.

Find

Cette méthode retourne l'index de la première ligne correspondant au critère de recherche. Si aucune ligne n'est trouvée, cette méthode retourne -1. Elle attend comme paramètre la valeur recherchée. Cette valeur est recherchée dans le champ utilisé comme critère de tri. Si le critère de tri est composé de plusieurs champs, il faut passer à la méthode `Find` un tableau d'objets contenant les valeurs recherchées pour chaque champ du critère de tri dans l'ordre d'apparition dans la propriété `Sort`.

Cette méthode est souvent utilisée pour rechercher une ligne à partir de la clé primaire.

```
Imports System.Data.SqlClient
Module TestFind
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim table As DataTable
    Dim ligne As DataRowView
    Dim codeClient As String
    Dim index As Integer

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true”
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT * from Customers”
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        Console.Write("saisir le code client : ")
        codeClient = Console.ReadLine()
        table.DefaultView.Sort = "CustomerID ASC"
        index = table.DefaultView.Find(codeClient)
        If index = -1 Then
            Console.WriteLine("il n'y a pas de client avec ce code")
        Else
            Console.WriteLine("le code {0} correspond au client {1}",
codeClient, table.DefaultView(index)("ContactName"))
        End If
        Console.ReadLine()
    End Sub
End Module
```

FindRows

Cette méthode recherche toutes les lignes correspondant au critère de recherche et retourne ces lignes sous forme d'un tableau de `DataRowView`.

Le code suivant recherche tous les clients d'un pays et d'une ville donnés :

```
Imports System.Data.SqlClient
Module TestFindRows
    Dim cmd As SqlCommand
```

```
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim table As DataTable
    Dim ligne As DataRowView
    Dim pays, ville As String
    Dim lignesTrouvees() As DataRowView
    Dim criteres() As Object

    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT * from Customers"
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        Console.Write("saisir le Pays : ")
        pays = Console.ReadLine()
        Console.Write("saisir la ville: ")
        ville = Console.ReadLine()
        table.DefaultView.Sort = "Country ASC,City ASC"
        criteres = (New Object() {pays, ville})
        lignesTrouvees = table.DefaultView.FindRows(criteres)
        If lignesTrouvees.Length = 0 Then
            Console.WriteLine("il n'y a pas de client correspondant")
        Else
            Console.WriteLine("les clients suivants correspondent ")
            For Each ligne In lignesTrouvees
                Console.WriteLine("Nom :{0}", ligne("ContactName"))
            Next
        End If
        Console.ReadLine()
    End Sub
End Module
```

4. Mettre à jour la base de données

Tout le travail effectué sur les données avec les méthodes vues précédemment est irrémédiablement perdu à la fermeture de l'application, si nous ne prenons pas soin de sauvegarder les données.

Dans la majorité des cas, les données proviennent d'une base de données, il faut donc la mettre à jour avec les modifications contenues dans un `DataSet`, une `DataTable` ou une `DataRow`. Le `DataAdapter` a été utilisé pour remplir le `DataSet`, c'est également à lui que l'on va faire appel pour mettre à jour la base de données.

Comme la méthode `Fill`, la méthode `Update` va utiliser des instructions SQL pour le dialogue avec la base de données. En fonction des besoins, elle utilisera l'instruction contenue dans la commande `InsertCommand`, `UpdateCommand` ou `DeleteCommand`. Si la méthode `Update` a besoin d'une commande et qu'elle n'est pas disponible, alors une exception est générée. La méthode `Fill` parcourt les lignes de la `DataTable` qu'elle doit mettre à jour et, en fonction de l'état de la ligne (`Added`, `Deleted`, `Modified`), appelle la commande `InsertCommand`, `DeleteCommand`, `UpdateCommand`. L'ordre dans lequel les mises à jour sont effectuées dans la base, peut avoir de l'importance. Pour contrôler l'ordre d'exécution des insertions, modifications et suppressions, vous pouvez procéder en trois étapes, en ne proposant à la méthode `Update` qu'un jeu restreint de lignes à mettre à jour. Vous pouvez, par exemple, ne sélectionner que les lignes effacées et demander la mise à jour de la base avec cet ensemble de lignes puis procéder de même avec les lignes modifiées et les lignes ajoutées.

La méthode `Select` permet d'obtenir un tableau de `DataRow` correspondant à un critère spécifique. C'est ce tableau de `DataRow` qui est passé comme paramètre à la méthode `Update`.

L'exemple suivant réalise les suppressions, les modifications puis les ajouts dans la base de données.

```
       Dim lignes() As DataRow
       ' recupere les lignes supprimees et demande la mise a jour de la base
       lignes = table.Select(Nothing, Nothing, DataViewRowState.Deleted)
       da.Update(table)
```

```
        ‘ recupere les lignes modifiees et demande la mise a jour de la base
        lignes = table.Select(Nothing, Nothing, DataViewRowState.ModifiedCurrent)
        da.Update(table)
        ‘ recupere les lignes ajoutees et demande la mise a jour de la base
        lignes = table.Select(Nothing, Nothing, DataViewRowState.Added)
        da.Update(table)
```

Cet exemple suppose bien sûr que les commandes `InsertCommand`, `DeleteCommand`, `UpdateCommand` soient définies au préalable.

a. Génération automatique de commandes

Les commandes chargées de la mise à jour de la base peuvent être générées automatiquement par un objet `SqlCommandBuilder`. Pour fonctionner correctement, le `SqlCommandBuilder` a quelques exigences :

- La propriété `SelectCommand` doit être définie pour le `DataAdapter` car c’est à partir de cette instruction SQL qu’il va générer les instructions `INSERT`, `UPDATE`, `DELETE`.
- La clé primaire doit être disponible dans la `DataTable`.
- Les données ne doivent pas provenir d’une jointure entre plusieurs tables.

Si une ou plusieurs de ces exigences ne sont pas respectées, il y a déclenchement d’une exception lors de la génération des commandes.

Les commandes sont générées en respectant les critères suivants :

InsertCommand

Insère une ligne dans la base pour toutes les lignes dont l’état est `Added`. Tous les champs, hormis les champs identité, expression ou TimeStamp sont mis à jour.

UpdateCommand

Met à jour dans la base toutes les lignes dont l’état est `Modified`. Tous les champs sont mis à jour sauf les champs identité, expression ou TimeStamp. La ligne à mettre à jour est recherchée dans la base, par la clé primaire, mais il faut également que les valeurs des autres champs dans la base correspondent à la version `Original` du champ dans la `DataRow`.

DeleteCommand

Efface de la base toutes les lignes dont l’état est `Deleted`. Il faut également que les valeurs présentes dans la base correspondent à la version `Original` des champs dans la `DataRow`. Les commandes générées sont disponibles via les méthodes `GetInsertCommand`, `GetUpdateCommand`, `GetDeleteCommand`.

L’exemple suivant affiche les instructions SQL des trois commandes générées automatiquement pour la table **Customers** :

```
Imports System.Data.SqlClient
Module TestOrdreMAJBase
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim table As DataTable
    Dim ligne As DataRowView
    Dim codeClient As String
    Dim index As Integer
    Dim bldr As SqlCommandBuilder
    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
```

```
        cmd.Connection = ctn
        cmd.CommandText = " SELECT * from Customers"
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        bldr = New SqlCommandBuilder(da)
        Console.WriteLine("Instruction SQL de UpadteCommand : {0}",
bldr.GetUpdateCommand.CommandText)
        Console.WriteLine("Instruction SQL de InsertCommand : {0}",
bldr.GetInsertCommand.CommandText)
        Console.WriteLine("Instruction SQL de DeleteCommand : {0}",
bldr.GetDeleteCommand.CommandText)
    End Sub
End Module
```

Ce code affiche les informations suivantes :

```
Instruction SQL de UpadteCommand : UPDATE [Customers] SET [CustomerID] = @p1,
[CompanyName] = @p2, [ContactName] = @p3, [ContactTitle] = @p4, [Address] = @p5,
[City] = @p6, [Region] = @p7, [PostalCode] = @p8, [Country] = @p9, [Phone] =
@p10, [Fax] = @p11 WHERE (([CustomerID] = @p12) AND ([CompanyName] = @p13) AND
((@p14 = 1 AND [ContactName] IS NULL) OR ([ContactName] = @p15)) AND ((@p16 =
1 AND [ContactTitle] IS NULL) OR ([ContactTitle] = @p17)) AND ((@p18 = 1 AND
[Address] IS NULL) OR ([Address] = @p19)) AND ((@p20 = 1 AND [City] IS NULL) OR
([City] = @p21)) AND ((@p22 = 1 AND [Region] IS NULL) OR ([Region] = @p23))
AND ((@p24 = 1 AND [PostalCode] IS NULL) OR ([PostalCode] = @p25))
AND ((@p26 = 1 AND [Country] IS NULL) OR ([Country] = @p27)) AND ((@p28 = 1 AND
[Phone] IS NULL) OR ([Phone] = @p29)) AND ((@p30 = 1 AND [Fax] IS NULL) OR
([Fax] = @p31)))
Instruction SQL de InsertCommand : INSERT INTO [Customers] ([CustomerID],
[CompanyName], [ContactName], [ContactTitle], [Address], [City], [Region],
[PostalCode], [Country], [Phone], [Fax]) VALUES (@p1, @p2, @p3, @p4, @p5, @p6,
@p7, @p8, @p9, @p10, @p11)
Instruction SQL de DeleteCommand : DELETE FROM [Customers] WHERE
(([CustomerID] = @p1) AND ([CompanyName] = @p2) AND ((@p3 = 1 AND [ContactName]
IS NULL) OR ([ContactName] = @p4)) AND ((@p5 = 1 AND [ContactTitle] IS NULL) OR
([ContactTitle] = @p6)) AND ((@p7 = 1 AND [Address] IS NULL) OR ([Address] =
@p8)) AND ((@p9 = 1 AND [City] IS NULL) OR ([City] = @p10)) AND ((@p11 = 1 AND
[Region] IS NULL) OR ([Region] = @p12)) AND ((@p13 = 1 AND [PostalCode]
IS NULL)OR ([PostalCode] = @p14)) AND ((@p15 = 1 AND [Country] IS NULL)
OR ([Country] = @p16)) AND ((@p17 = 1 AND [Phone] IS NULL) OR ([Phone] =
@p18)) AND ((@p19 = 1 AND [Fax] IS NULL) OR ([Fax] = @p20)))
```

Le moins que l'on puisse dire, c'est que ce code n'est pas très parlant !

Rassurez-vous dans le paragraphe sur les accès concurrents, nous allons éclaircir la présence de ces innombrables paramètres dans ces trois instructions SQL. L'important, pour le moment, est que ces instructions réalisent correctement la mise à jour de la base de données.

Nous allons le vérifier en réalisant un ajout, une modification et une suppression dans la table **Customers** sur des clients Français. Regardons l'état de la table avant d'effectuer nos modifications.

```
select * from customers where country='France'
```

	CustomerID	CompanyName	ContactName	ContactTitle	Address
1	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber
2	BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers
3	DUMON	Du monde entier	Janine Labrune	Owner	67, rue des Cinquante Otages
4	FOLIG	Folies gourmandes	Martine Rancé	Assistant Sales Agent	184, chaussée de Tournai
5	FRANR	France restauration	Carine Schmitt	Marketing Manager	54, rue Royale
6	LACOR	La corne d'abondance	Daniel Tonini	Sales Representative	67, avenue de l'Europe
7	LAMAI	La maison d'Asie	Annette Roulet	Sales Manager	1 rue Alsace-Lorraine
8	PARIS	Paris spécialités	Marie Bertrand	Owner	265, boulevard Charonne
9	SPECD	Spécialités du monde	Dominique Perrier	Marketing Manager	25, rue Lauriston
10	VICTE	Victuailles en stock	Mary Saveley	Sales Agent	2, rue du Commerce
11	VINET	Vins et alcools Chevalier	Paul Henriot	Accounting Manager	59 rue de l'Abbaye

Exécutons ensuite le code ci-dessous :

```
Imports System.Data.SqlClient
Module TestMAJBase
    Dim cmd As SqlCommand
    Dim ctn As SqlConnection
    Dim ds As DataSet
    Dim da As SqlDataAdapter
    Dim table As DataTable
    Dim ligne As DataRow
    Dim bldr As SqlCommandBuilder
    Public Sub main()
        ctn = New SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        cmd = New SqlCommand
        cmd.Connection = ctn
        cmd.CommandText = " SELECT * from Customers where country='France'"
        ds = New DataSet
        da = New SqlDataAdapter(cmd)
        da.Fill(ds, "Clients")
        table = ds.Tables("Clients")
        'efface Frederique Citeaux
        table.Rows(7).Delete()
        ' change adresse Carine Schmitt
        table.Rows(4)("Address") = "9 rue Benjamin Franklin"
        ' ajout d'un nouveau client
        ligne = table.NewRow
        ligne("CustomerID") = "ENIEC"
        ligne("CompanyName") = "Eni Ecole Informatique"
        ligne("ContactName") = "Marcel dupond"
        ligne("ContactTitle") = "Directeur"
        ligne("Address") = "24 rue Crébilon"
        ligne("Country") = "France"
        ligne(_City")="Nantes"
        table.Rows.Add(ligne)
        bldr = New SqlCommandBuilder(da)
        da.Update(table)
    End Sub
End Module
```

Comparons le contenu actuel avec le contenu précédent de la table **Customers**, pour vérifier que les modifications ont bien été prises en compte.

```
select * from customers where country='France'
```

	CustomerID	CompanyName	ContactName	ContactTitle	Address	City
1	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbo
2	BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseil
3	DUMON	Du monde entier	Janine Labrune	Owner	67, rue des Cinquante Otages	Nantes
4	ENIEC	Eni Ecole Informatique	Marcel dupond	Directeur	24 rue Crébilon	Nantes
5	FOLIG	Folies gourmandes	Martine Rancé	Assistant Sales Agent	184, chaussée de Tournai	Lille
6	FRANR	France restauration	Carine Schmitt	Marketing Manager	9 rue Benjamin Franklin	Nantes
7	LACOR	La corne d'abondance	Daniel Tonini	Sales Representative	67, avenue de l'Europe	Versail
8	LAMAI	La maison d'Asie	Annette Roulet	Sales Manager	1 rue Alsace-Lorraine	Toulous
9	SPECD	Spécialités du monde	Dominique Perrier	Marketing Manager	25, rue Lauriston	Paris
10	VICTE	Victuailles en stock	Mary Saveley	Sales Agent	2, rue du Commerce	Lyon
11	VINET	Vins et alcools Chevalier	Paul Henriot	Accounting Manager	59 rue de l'Abbaye	Reims

b. Utilisation de commandes personnalisées

L'utilisation de commandes personnalisées permet de choisir le type d'action effectué lors de la mise à jour de la base de données. Par exemple, l'effacement d'une ligne peut se traduire par l'affectation d'une valeur particulière à un champ de l'enregistrement. Dans ce cas, l'instruction **SQL** exécutée dans la `DeleteCommand` sera une instruction `UPDATE` plutôt qu'une instruction `DELETE`.

Par exemple, le code suivant crée une commande personnalisée pour la suppression :

```
delCmd = New SqlCommand
 delCmd.Connection = ctn
 delCmd.CommandText = " UPDATE Customers set Archive=1 where CustomerID=@num"
 codeClient = New SqlParameter("@num", SqlDbType.NChar, 5, ParameterDirection.
Input, False, Nothing, Nothing, "CustomerID", DataRowVersion.Current, Nothing)
 delCmd.Parameters.Add(codeClient)
 da.DeleteCommand = delCmd
 bldr = New SqlCommandBuilder(da)
```

Les commandes personnalisées sont compatibles avec les commandes générées automatiquement par le `SqlCommandBuilder` puisque celui-ci ne génère une commande que si la propriété `InsertCommand`, `DeleteCommand` ou `UpdateCommand` est égale à **nothing** dans le `DataAdapter`. Si une commande existe, elle n'est pas remplacée par le `SqlCommandBuilder`.

c. Gestion des accès concurrents

Dans un environnement multiutilisateurs, deux techniques existent pour la gestion des mises à jour : le verrouillage optimiste et le verrouillage pessimiste.

Le verrouillage pessimiste est le plus contraignant pour les utilisateurs puisque, pour éviter les conflits lors des modifications de la base, dès qu'un utilisateur souhaite modifier un enregistrement, celui-ci est verrouillé dans la base. Tant que la modification de l'enregistrement n'est pas terminée, le verrou reste actif, bloquant ainsi l'accès des autres utilisateurs. L'optique de cette solution est d'éviter l'apparition de conflits.

Le verrouillage optimiste n'est pas réellement un verrouillage puisque les enregistrements sont pratiquement disponibles en permanence. C'est au moment de la mise à jour qu'un test est effectué, pour vérifier si les données présentes dans la base sont identiques aux données utilisées pour remplir le `DataSet`. Si les données sont différentes, c'est qu'un autre utilisateur les a modifiées entre le remplissage du `DataSet` et la demande de mise à jour de la base. Il convient alors de prendre une décision concernant les mises à jour.

Trois solutions sont envisageables :

- On abandonne les mises à jour.
- On écrase la version existante.
- On demande à l'utilisateur ce qu'il souhaite.

Cette solution peut être mise en œuvre avec deux techniques courantes.

La première est d'utiliser, dans la table, un champ de type **TimeStamp**. La particularité de ce type de champ est d'avoir une valeur changeant automatiquement à chaque modification effectuée sur l'enregistrement. Il suffit donc d'effectuer une comparaison de la valeur présente dans la base avec la valeur présente dans l'application. S'il y a une différence, c'est que la base a été modifiée depuis le chargement du `DataSet`.

Pour que cette solution soit envisageable, il faut que la base de données soit capable de gérer ce type de champ. Cette solution augmente aussi le volume des données puisque, par exemple pour SQL Server, ce type de champ utilise huit octets.

La deuxième solution est de gérer cela au niveau de l'application, en conservant les données originales et en les comparant avec les données présentes dans la base au moment de la mise à jour. C'est cette solution qui est utilisée dans les commandes générées automatiquement par l'objet`SqlCommandBuilder`.

Analysons le code d'une requête de modification générée par cet objet.

```
UPDATE [Customers] SET [CustomerID] = @p1, [CompanyName] = @p2, [ContactName] =
@p3, [ContactTitle] = @p4, [Address] = @p5, [City] = @p6, [Region] = @p7,
[PostalCode] = @p8, [Country] = @p9, [Phone] = @p10, [Fax] = @p11 WHERE
(([CustomerID] = @p12) AND ([CompanyName] = @p13) AND ((@p14 = 1 AND
[ContactName] IS NULL) OR ([ContactName] = @p15)) AND ((@p16 = 1 AND
[ContactTitle] IS NULL) OR ([ContactTitle] = @p17)) AND ((@p18 = 1 AND
[Address]IS NULL) OR ([Address] = @p19)) AND ((@p20 = 1 AND [City] IS NULL) OR
([City] = @p21)) AND ((@p22 = 1 AND [Region] IS NULL) OR ([Region] =
@p23)) AND ((@p24 = 1AND [PostalCode] IS NULL) OR ([PostalCode] =
@p25)) AND ((@p26 = 1 AND [Country]IS NULL) OR ([Country] =
@p27)) AND ((@p28 = 1 AND [Phone] IS NULL) OR<+>([Phone] = @p29)) AND ((@p30 = 1
AND [Fax] IS NULL) OR ([Fax] = @p31)))
```

Cette commande utilise un nombre impressionnant de paramètres : trente et un en fait. Essayons d'expliquer le rôle de chacun de ces paramètres.

Les paramètres de @p1 à @p11 sont utilisés pour spécifier les nouvelles valeurs de l'enregistrement.

Le paramètre @p12 est utilisé pour identifier l'enregistrement à mettre à jour en effectuant un test sur la clé primaire.

Les autres paramètres ne sont là que pour la gestion du verrouillage optimiste. Pour chaque champ, on vérifie que les données présentes dans la base sont identiques aux données du `DataSet`. Pour les champs n'autorisant pas les valeurs nulles, la vérification est très simple. C'est le cas du champ `CompanyName` qui est comparé avec le paramètre @p13.

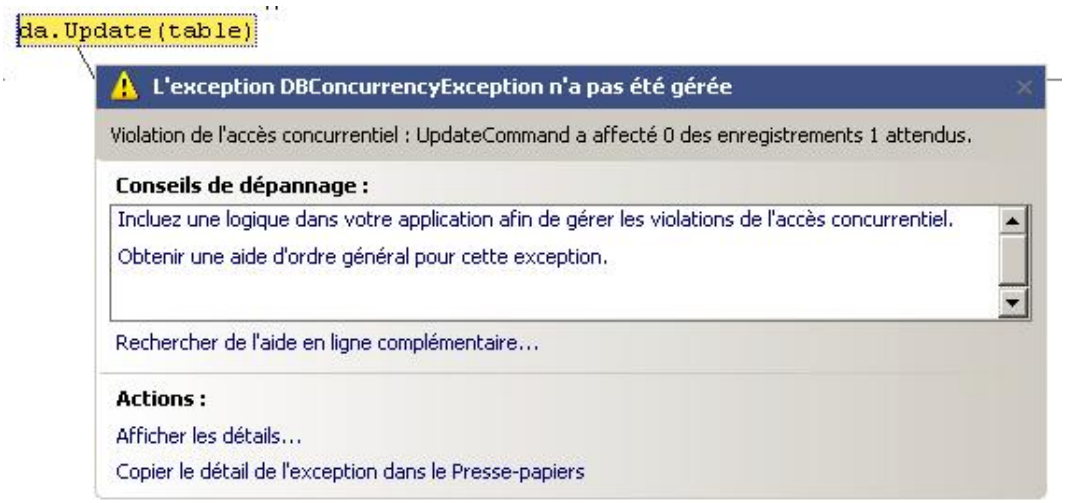
Pour les champs autorisant les valeurs nulles, il faut utiliser une petite astuce. En effet, il est seulement possible de vérifier qu'un champ est nul ou pas dans la base, mais il ne peut pas y avoir de comparaison avec la valeur **Null**. Pour pallier cette limitation, les paramètres @p14,@p16,@p18,@p20,@p22 @p24,@p26,@p28,@p30 sont utilisés pour représenter la valeur **Null** pour un champ dans le `DataSet`. Si leur valeur est égale à 1, c'est que le champ est égal à **Null** dans le `DataSet`. Le test sur la valeur de ces paramètres est donc combiné avec l'opérateur `And` pour vérifier la nullité simultanée du champ correspondant dans la base. Lors de l'exécution de la commande, les paramètres sont remplacés par les valeurs présentes dans le `DataSet`. En fonction du paramètre, des versions différentes de la ligne sont utilisées.

Pour les paramètres @p1 à @p11, les versions `Current` sont utilisées. Ce sont les données que l'on veut transférer dans la base de données. Pour les autres paramètres, ce sont les versions `Original` des paramètres qui sont utilisées.

La version de la ligne à utiliser est déterminée au moment de la création du paramètre, avant son ajout dans la collection **Parameters**. Pour cela, le constructeur suivant est utilisé :

```
Public Sub New (parameterName As String, dbType As SqlDbType, ,size As Integer,
direction As ParameterDirection, isNullable As Boolean, precision As Byte,
scaleAs Byte, sourceColumn As String, sourceVersion As DataRowVersion, value As
Object )
```

Il permet d'associer automatiquement une version d'un champ particulier à un paramètre. C'est le rôle des arguments `sourceColumn` et `sourceVersion`. L'exécution de la commande renvoie le nombre d'enregistrements réellement mis à jour. Si la commande ne peut pas mettre à jour les champs, alors la valeur renvoyée par l'exécution de la commande est égale à zéro. Dans ce cas, il y a déclenchement d'une exception :



Une solution moins brutale permet de surveiller les mises à jour au fur et à mesure de l'exécution des instructions SQL. Il faut pour cela gérer l'événement `RowUpdated` du `DataAdapter` qui est déclenché après l'appel de chaque `InsertCommand`, `DeleteCommand`, `UpdateCommand`. Le paramètre de type **RowUpdatedEventArgs** permet de savoir par la propriété `RecordsAffected` combien d'enregistrements ont été mis à jour.

Si aucun enregistrement n'a été mis à jour, vous pouvez choisir l'action à entreprendre en modifiant la propriété `Status` avec l'une des valeurs de l'énumération `UpdateStatus` :

Continue

La mise à jour se poursuit comme si rien ne s'était passé.

ErrorsOccurred

Une exception va être déclenchée.

SkipAllRemainingRows

Arrêt de la mise à jour des lignes restantes.

SkipCurrentRow

Arrêt de la mise à jour de la ligne courante. La mise à jour continue avec les lignes restantes.

Le type d'action à entreprendre doit, en principe, être laissé à l'appréciation de l'utilisateur comme dans l'exemple suivant :

```
Private Sub da_RowUpdated(ByVal sender As Object, ByVal e As System.Data.
SqlClient.SqlRowUpdatedEventArgs) Handles da.RowUpdated
        Dim reponse As Integer
        If e.RecordsAffected = 0 Then
            Console.WriteLine("Une erreur s'est produite lors
de la mise à jour de la base de données")
            Console.WriteLine("vous souhaitez :")
            Console.WriteLine("1 - continuer les mises a jour")
            Console.WriteLine("2 - annuler les mises a jour")
            Console.WriteLine("3 - annuler la mise a jour de cette ligne mais
continuer pour les autres lignes")
            reponse = Console.ReadLine()
            Select Case reponse
                Case 1
                    e.Status = UpdateStatus.Continue
                Case 2
                    e.Status = UpdateStatus.SkipAllRemainingRows
                Case 3
                    e.Status = UpdateStatus.SkipCurrentRow
            End Select
        End If
    End Sub
```

5. Les transactions

Les transactions permettent de regrouper dans une entité, un ensemble de commandes SQL. Ce regroupement va garantir que, si l'une des instructions contenues dans la transaction échoue, la base de données pourra retrouver son état initial. L'exemple classique est le virement d'un compte bancaire à un autre. Imaginez que vous ayez, dans votre base de données, une table pour les comptes des particuliers et une table pour les comptes des entreprises.

Le transfert d'un compte entreprise vers un compte particulier (le paiement de votre salaire) pourrait s'effectuer avec les instructions suivantes :

```
Imports System.Data.SqlClient
Module TestTransaction
    Dim cmdPart, cmdEnt As SqlCommand
    Dim ctn As SqlConnection
    Dim numParticulier, numEntreprise, montantPart, montantEnt As SqlParameter
    Public Sub main()
        ctn = New SqlClient.SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        cmdEnt = New SqlClient.SqlCommand
        cmdEnt.Connection = ctn
        cmdEnt.CommandText = "Update ComptesEntreprise set solde=
solde-@montant where numCompte=@numCompte"
        numEntreprise = New SqlParameter("@numCompte", SqlDbType.Int)
        numEntreprise.Value = 1234
        cmdEnt.Parameters.Add(numEntreprise)
        montantEnt = New SqlParameter("@montant", SqlDbType.Decimal)
        montantEnt.Value = 3000
        cmdEnt.Parameters.Add(montantEnt)
        cmdEnt.ExecuteNonQuery()
        cmdPart = New SqlClient.SqlCommand
```

```
        cmdPart.Connection = ctn
        cmdPart.CommandText = "Update ComptesParticulier set solde=
solde+@montant  where numCompte=@numCompte"
        numParticulier = New SqlParameter("@numCompte", SqlDbType.Int)
        numParticulier.Value = 5678
        cmdPart.Parameters.Add(numParticulier)
        montantPart = New SqlParameter("@montant", SqlDbType.Decimal)
        montantPart.Value = 3000
        cmdPart.Parameters.Add(montantPart)
        cmdPart.ExecuteNonQuery()
        ctn.Close()
    End Sub

End Module
```

Que se passe-t-il, si pendant l'exécution de ce code le serveur de base de données devient indisponible à cause d'un problème réseau par exemple ? L'opération de débit peut avoir été effectuée alors que l'opération de crédit n'a pas pu s'exécuter correctement. Il risque donc d'y avoir un gros problème dans le fonctionnement de l'application (et pour le paiement de votre salaire). Les transactions vont permettre de résoudre ce problème, en regroupant l'exécution d'instructions SQL pour garantir qu'elles seront toutes exécutées ou qu'aucune ne sera exécutée.

Les transactions sont gérées au niveau de la connexion, c'est donc elle qui va nous permettre de démarrer une transaction. La méthode `BeginTransaction` nous retourne une instance de la classe `SqlTransaction`. Pour chaque exécution de commande, nous pouvons alors indiquer si l'exécution doit se passer dans le contexte de la transaction ou à l'extérieur. À la fin du traitement, nous pouvons valider toutes les instructions confiées à la transaction ou au contraire les annuler toutes. La méthode `Commit` valide la transaction alors que la méthode `RollBack` l'annule.

Pour sécuriser le code précédent, nous pourrions utiliser la version suivante :

```
Imports System.Data.SqlClient
Module TestTransaction
    Dim cmdPart, cmdEnt As SqlCommand
    Dim ctn As SqlConnection
    Dim numParticulier, numEntreprise, montantPart, montantEnt As SqlParameter
    Dim trans As SqlTransaction
    Public Sub main()
        ctn = New SqlClient.SqlConnection()
        ctn.ConnectionString = "Data Source=localhost;Initial
Catalog=Northwind;Integrated Security=true"
        ctn.Open()
        trans = ctn.BeginTransaction
        Try
            cmdEnt = New SqlClient.SqlCommand
            cmdEnt.Connection = ctn
            cmdEnt.CommandText = "Update ComptesEntreprise set solde=solde-
@montant where numCompte=@numCompte"
            numEntreprise = New SqlParameter("@numCompte", SqlDbType.Int)
            numEntreprise.Value = 1234
            cmdEnt.Parameters.Add(numEntreprise)
            montantEnt = New SqlParameter("@montant", SqlDbType.Decimal)
            montantEnt.Value = 3000
            cmdEnt.Parameters.Add(montantEnt)
            ' place l'execution de la commande dans la transaction
            cmdEnt.Transaction = trans
            cmdEnt.ExecuteNonQuery()
            cmdPart = New SqlClient.SqlCommand
            cmdPart.Connection = ctn
            cmdPart.CommandText = "Update ComptesParticuliers set solde=solde+
@montant where numCompte=@numCompte"
            numParticulier = New SqlParameter("@numCompte", SqlDbType.Int)
            numParticulier.Value = 5678
            cmdPart.Parameters.Add(numParticulier)
            montantPart = New SqlParameter("@montant", SqlDbType.Decimal)
            montantPart.Value = 3000
            cmdPart.Parameters.Add(montantPart)
            cmdPart.Transaction = trans
            cmdPart.ExecuteNonQuery()
            trans.Commit()
        Catch ex As Exception
            trans.Rollback()
```

```
            Console.WriteLine("toutes les opérations ont ete annulees")
        End Try
        ctn.Close()
    End Sub
End Module
```

Si la connexion est interrompue, l'instruction `RollBack` ou `Commit` ne pourra pas être acheminée vers le serveur. Dans ce cas, le serveur prend l'initiative d'exécuter un `RollBack` sur toutes les transactions en cours, si la connexion avec le client est perdue.

Présentation de LINQ

Après de nombreuses années d'évolution les langages objets sont devenus incontournables dans les développements informatiques. Parallèlement les systèmes de stockage ont également évolués principalement sur deux axes : les bases de données et les fichiers XML. La cohabitation entre concepts objets et données se fait tant bien que mal par l'ajout aux langages objets de quelques capacités pour dialoguer avec les données. Cette solution n'est pas entièrement satisfaisante car elle présente les inconvénients suivants :

- Le langage utilisé pour manipuler les données est très souvent spécifique à un type de source de données.
- Les mots clés de ce langage sont inconnus du langage de programmation qui les considère comme de simples chaînes de caractères donc pas de vérification syntaxique avant l'exécution.
- Le changement de type de source de données entraîne de lourdes modifications du code et une nouvelle période d'apprentissage pour le développeur.
- Les types de données sont parfois incompatibles entre le langage de programmation et la source de données. Il faut dans ce cas réaliser des conversions souvent gourmandes en temps et même parfois dangereuses.

Avec LINQ tout ceci va devenir que des mauvais souvenirs. Mais que se cache-t-il derrière ces quatre lettres : *Language Integrated Query* ou Langage de requête intégré ? Il s'agit donc d'un langage de requête permettant l'interrogation de sources de données. Mais qu'a-t-il de plus que ce brave SQL ? La clé du mystère se situe dans le terme 'intégré'. En effet contrairement à d'autres méthodes utilisées pour interroger des sources de données (SQL, XPATH…), LINQ fait partie du langage dans lequel l'application est développée (VB, C#…). Un autre point très important concernant LINQ réside dans la syntaxe même du langage. Celle-ci sera identique quel que soit le type de source de données interrogée : tableau, collection, base de données, fichier XML, dataset… Le dernier point important de cette présentation de LINQ concerne les données manipulées. Votre application est développée avec un langage objet et bien LINQ lui aussi manipule des objets. Il n'y a donc pas besoin de réaliser manuellement les opérations de conversion. Si elles sont nécessaires, elles seront réalisées automatiquement par LINQ. Après ce bref aperçu, regardons maintenant la syntaxe de LINQ.

Syntaxe du langage LINQ

Avant de détailler la syntaxe de LINQ, nous allons étudier un premier exemple très simple. Dans les exemples de ce paragraphe, nous utiliserons comme source de données deux listes remplies avec des instances des classes du diagramme ci-après.

Les données utilisées pour créer ces instances de classe sont extraites de la base de données Northwind. Elles sont placées dans un fichier texte qui est ensuite lu par le code pour recréer les instances de classe en mémoire. Voici l'extrait de code permettant de réaliser ces opérations.

```
Dim listeCommandes As List(Of Commande)
Dim listeClients As List(Of Client)
Sub Main()
    listeCommandes = New List(Of Commande)
    listeClients = New List(Of Client)
    Dim co As Commande
    Dim cl As Client
    Dim f As IO.StreamReader
    Dim ligne As String
    Dim col As String()
    Dim nom As String = ""
    f = New IO.StreamReader("c:\data.txt", IO.FileMode.Open)
    Do
      ligne = f.ReadLine
      If Not IsNothing(ligne) Then
          col = ligne.Split(New Char() {vbTab})
          If nom <> col(0) Then
             nom = col(0)
             cl = New Client() With {.nom = col(0), .adresseFacture =
New Adresse With {.rue = col(1), .ville = col(2), .codePostal = col(3)},
.telephone = col(4)}
          listeClients.Add(cl)
          End If
          co = New Commande() With {.codeCommande = col(5), .dateCommande =
col(6), .port = col(7), .adresseLivraison = New Adresse With {.rue = col(8),
.ville = col(0), .codePostal = col(10)}}
          listeCommandes.Add(co)
          co.LeClient = cl
```

```
            cl.LesCommandes.Add(co)
        End If
    Loop Until ligne Is Nothing
    f.Close()
End Sub
```

Dans les exemples qui suivent, nous supposerons que cette portion de code a déjà été exécutée pour remplir les deux listes. Cette ébauche de projet est disponible en téléchargement sur le site de l'éditeur.

1. Premières requêtes LINQ

Une requête LINQ est constituée de trois actions :

- l'obtention des données ;
- la création de la requête elle-même ;
- l'exécution de la requête.

La première étape est très simple puisque pour pouvoir être utilisée comme source de données, une classe doit simplement implémenter l'interface générique `IEnumerable(T)`. C'est le cas de nombreuses classes du Framework .NET qui sont donc directement utilisables dans des requêtes LINQ.

Pour les sources de données n'implémentant pas cette interface, comme par exemple un document XML, des classes utilitaires permettent de les rendre compatibles avec LINQ.

Le plus gros du travail est constitué par la deuxième étape : la création de la requête elle-même.

Dans la requête, nous allons indiquer quelles informations nous souhaitons obtenir de la source de données, comment elles seront triées, regroupées ou structurées.

La requête contient trois clauses :

- `From` : indiquant l'origine des données.
- `Where` : spécifie la ou les conditions pour que les données soient comprises dans les valeurs retournées.
- `Select` : indique quelles sont les informations retournées à partir de la source de données.

Voici donc ci-dessous notre première requête LINQ.

```
Dim requete = From unClient In listeClients _
              Where unClient.nom Like "A*" _
              Select unClient
```

Généralement une requête LINQ est stockée dans une variable puis est exécutée ultérieurement. Il est important de bien se souvenir que la variable contenant la requête n'exécute aucune action et ne retourne aucune donnée. Elle stocke simplement la définition de la requête. L'exécution de la requête n'a lieu que lorsque l'on s'intéresse aux données qu'elle retourne. C'est le cas dans l'exemple suivant où nous parcourons les résultats dans une boucle `For Each`.

```
For Each unClient In requete
    Console.WriteLine(unClient.nom)
Next
```

Ce mécanisme permet d'exécuter plusieurs fois la même requête sans être obligé de la redéfinir à chaque exécution.

Cependant, dans certains cas, la variable contient le résultat de la requête et non la requête elle-même. C'est le cas par exemple, lorsque qu'il y a un calcul d'agrégat dans la requête. Dans l'exemple suivant, nous recherchons combien il y a de clients dont le nom commence par la lettre 'A'. Le résultat de la requête est dans ce cas un simple entier qui est calculé dès la définition de la requête.

```
Dim nbClients = (From unClient In listeClients _
                    Where unClient.nom Like "A*" _
                    Select unClient).Count
Console.WriteLine(nbClients)
```

Vous pouvez également utiliser plusieurs sources de données dans la clause `From`. Pour cela, le mot clé `Join` permet de combiner les données des différentes sources. L'exemple suivant recherche les clients dont le nom commence par un 'A' et pour chacun récupère la date de toutes les commandes.

```
Dim requete3 =From unClient In listeClients _
              Join uneCommande In listeCommandes _
             On unClient.nom Equals uneCommande.LeClient.nom _
             Where unClient.nom Like "A*" _
Select nomCli = unClient.nom, _
 dateCde = uneCommande.dateCommande

For Each r In requete3
    Console.WriteLine(r.nomCli & "  " & r.dateCde)
Next
```

Ce code mérite un petit commentaire supplémentaire concernant la clause`Select`. Nous souhaitons obtenir le nom du client et la date des commandes soit une chaîne de caractères et une date. Il est inutile d'utiliser une instance de la classe `Client` et une instance de la classe `Commande` pour simplement ces deux informations. Le compilateur génère donc une classe anonyme pour ces deux informations avec une propriété `nomCli` et une propriété `dateCde`. Le type de la variable `r` utilisée dans la boucle `For Each` pour parcourir le résultat de la requête sera déterminé implicitement pour correspondre au type anonyme créé par le compilateur.

2. Les opérateurs de requête

Les opérateurs permettant la création de requête LINQ peuvent être classés en huit catégories :

- Tri de données
- Opérations sur des ensembles de données
- Filtrage
- Projection
- Partitionnement
- Jointures, regroupements
- Quantificateurs
- Agrégation

Pour écrire des requêtes LINQ efficaces, il convient de bien connaître ces opérateurs. Nous allons donc les détailler avec de nombreux exemples.

a. Tride données

Il très facile d'obtenir les résultats d'une requête triés selon un ou plusieurs critères. Grâce à l'opérateur `Order By`, nous devons indiquer la propriété sur laquelle va être réalisé le tri. La requête suivante trie les clients en fonction de leur nombre de commandes.

```
Dim requeteTri1 = From unclient In listeClients _
                        Order By unclient.LesCommandes.Count _
                        Select unclient
        For Each unClient In requeteTri1
            Console.WriteLine(unClient.nom & "nb commandes:"
& unClient.LesCommandes.Count)
        Next
```

Par défaut le tri se fait par ordre croissant. Pour obtenir les meilleurs clients en début de liste, il est préférable d'utiliser le mot clé `Descending`à la suite du critère de tri.

```
Dim requeteTri2 = From unclient In listeClients _
                        Order By unclient.LesCommandes.Count Descending _
                        Select unclient
        For Each unClient In requeteTri2
            Console.WriteLine(unClient.nom & "nb commandes:"
& unClient.LesCommandes.Count)
        Next
```

Plusieurs critères de tri peuvent être indiqués pour lever les ambiguïtés lorsque deux propriétés ont la même valeur. Les critères de tri doivent être séparés par des virgules dans la requête. La requête trie les clients sur le nombre de commandes dans l'ordre décroissant puis sur le nom du client dans l'ordre croissant en cas d'égalité du nombre de commandes.

```
Dim requeteTri3 = From unclient In listeClients _
                        Order By unclient.LesCommandes.Count Descending,
unclient.nom Ascending _
                        Select unclient
        For Each unClient In requeteTri3
            Console.WriteLine(unClient.nom & "nb commandes:"
& unClient.LesCommandes.Count)
        Next
```

b. Opérations sur des ensembles de données

Le seul opérateur disponible dans cette catégorie permet l'élimination des doublons lors de la recherche d'informations. Le mot clé`Distinct` placé à la fin de la clause `Select` indique que les doublons seront éliminés. L'ensemble des éléments de la clause `Select` est pris en compte pour l'élimination des doublons. La requête suivante détermine les différentes villes où nous avons des clients. Si nous avons plusieurs clients dans la même ville, celle-ci ne sera listée qu'une seule fois.

```
Dim requeteEnsemble = From unclient In listeClients _
                            Order By unclient.adresseFacture.ville _
                            Select unclient.adresseFacture.ville Distinct
        For Each ville In requeteEnsemble
            Console.WriteLine(ville)
        Next
```

c. Filtrage de données

Le filtrage consiste à réduire le nombre d'éléments retournés par la requête. Une ou plusieurs expressions sont ajoutées à la requête à l'aide de la clause`Where`. Elles doivent fournir un booléen lors de l'évaluation de la requête. Les opérateurs de comparaison standard de Visual Basic peuvent être utilisés à l'intérieur de l'expression. L'utilisation de chaînes de caractères dans une clause `Where` mérite une petite précision. Bien que l'on puisse utiliser l'opérateur '=' pour un critère de filtrage portant sur une chaîne de caractères, l'utilisation de la méthode `Equals` offre beaucoup plus de fonctionnalités. En effet, avec l'opérateur '=' il doit y avoir une stricte égalité avec une distinction entre minuscules et majuscules pendant le test. La méthode `Equals` est plus souple puisque elle permet d'indiquer comment doit se faire la comparaison et éventuellement ignorer la distinction entre minuscules et majuscules comme dans l'exemple ci-dessous.

```
Dim requeteFiltrage = From unclient In listeClients _
Where unclient.adresseFacture.ville.Equals("nantes",
StringComparison.OrdinalIgnoreCase) _
Select unclient

        For Each unclient In requeteFiltrage
            Console.WriteLine(unclient.nom)
        Next
```

d. Projections

Une opération de projection correspond à la transformation d'un objet en une nouvelle forme. Cette nouvelle forme est constituée par l'ensemble des propriétés de l'objet spécifié dans la clause `Select`. En utilisant la projection, on peut automatiquement créer un nouveau type qui est construit à partir de chaque objet. Vous pouvez projeter une propriété directement ou bien exécuter une fonction prenant comme paramètre la propriété. C'est dans ce cas le

résultat de la fonction qui est utilisé comme valeur pour la propriété de l'objet créé. Vous pouvez aussi projeter l'objet original sans le modifier.

```
Dim requeteProjection = From unclient In listeClients _
Select nomCli = unclient.nom.ToUpper, villeCli =
unclient.adresseFacture.ville.ToLower

For Each r In requeteProjection
    Console.WriteLine(r.nomCli & " " & r.villeCli)
Next
```

Les projections sont également réalisables en indiquant plusieurs clauses`From` dans la requête. Dans ce cas, le résultat de la requête fait correspondre chaque objet de chacune des sources de données avec tous les objets des autres sources.

```
Dim requeteProjection1 = From unclient In listeClients _
                         From unCommande In listeCommandes _
                         Select cli = unclient, cmd = unCommande
        For Each r In requeteProjection1
            Console.WriteLine(r.cli.nom & " " & r.cmd.dateCommande)
        Next
```

Ce genre d'opération conduit très rapidement à une explosion combinatoire. En effet, le nombre d'objets dans le résultat de la requête est égal au produit des nombres d'objets dans chacune des sources de données. Un filtrage permettant de restreindre le nombre d'objets dans le résultat est en général souhaitable avec ce type de projection.

```
Dim requeteProjection2 = From unclient In listeClients _
                         From unCommande In listeCommandes _
                         Where unclient.Equals(unCommande.LeClient) _
                         Select cli = unclient, cmd = unCommande
        For Each r In requeteProjection2
            Console.WriteLine(r.cli.nom & " " & r.cmd.dateCommande)
        Next
```

e. Partitionnement

Le partitionnement consiste à découper en deux parties un ensemble de données et à retourner une des deux parties. La limite du découpage peut être absolue, et dans ce cas exprimée en nombre d'objets, ou conditionnelle. Deux clauses sont utilisées pour le partitionnement :

- `Skip`indique que l'on souhaite obtenir la deuxième partie de la liste (en fait on 'saute' les objets placés au début) ;
- `Take`indique que l'on souhaite obtenir le début de la liste sans tenir compte des enregistrements de la fin de liste.

Voyons comment utiliser ces deux opérateurs avec la syntaxe absolue et la syntaxe conditionnelle.

La requête suivante permet d'obtenir la liste des dix plus mauvais clients (en se basant sur le nombre de commandes).

```
Dim requetePartition = From unclient In listeClients _
                              Order By unclient.LesCommandes.Count _
                               Skip listeClients.Count - 10 _
                               Select unclient
        For Each unclient In requetePartition
            Console.WriteLine(unclient.nom)
        Next
```

Pour illustrer la syntaxe conditionnelle, nous recherchons maintenant tous les clients ayant un nombre de commandes inférieur ou égal à 5.

```
Dim requetePartition1 = From unclient In listeClients _
                        Order By unclient.LesCommandes.Count Descending _
                        Skip While unclient.LesCommandes.Count > 5 _
                        Select unclient
For Each unclient In requetePartition1
```

```
        Console.WriteLine(unclient.nom & " " & unclient.LesCommandes.Count)
Next
```

Pour récompenser nos clients fidèles, recherchons maintenant les dix meilleurs d'entre eux.

```
Dim requetePartition2 = From unclient In listeClients _
                        Order By unclient.LesCommandes.Count Descending _
                        Take 10 _
                        Select unclient
        For Each unclient In requetePartition2
            Console.WriteLine(unclient.nom)
        Next
```

Et enfin recherchons les clients ayant passés au moins dix commandes.

```
Dim requetePartition3 = From unclient In listeClients _
                         Order By unclient.LesCommandes.Count Descending _
                         Take While unclient.LesCommandes.Count >= 10 _
                         Select unclient
For Each unclient In requetePartition3
       Console.WriteLine(unclient.nom & " " & unclient.LesCommandes.Count)
Next
```

f. Jointures et regroupements

La jointure de deux sources de données correspond à l'association d'une des sources de données avec les objets de l'autre source de données ayant une propriété commune. En programmation objet, les jointures permettent de remplacer des associations incomplètes. Dans l'exemple que nous utilisons depuis le début de ce chapitre, la classe `Client` contient une propriété permettant d'obtenir la liste des commandes d'un client (LesCommandes) et la classe `Commande` contient un attribut permettant de référencer le client ayant passé la commande (LeClient). L'association est donc, dans notre cas, bidirectionnelle. Si par économie ou par oubli, la propriété `LesCommandes` de la classe `Client` n'existe pas, il faut dans ce cas parcourir la liste des commandes et tester chacune d'elles pour trouver toutes les commandes d'un client précis. C'est ce travail que réalise la jointure. La requête suivante obtient les commandes de chaque client.

```
Dim requeteJoin = From unclient In listeClients _
Join uneCommande In listeCommandes On unclient Equals uneCommande.LeClient _
Select unclient, uneCommande
For Each r In requeteJoin
     Console.WriteLine(r.unclient.nom & " " & r.uneCommande.dateCommande)
Next
```

Pour chaque commande les informations concernant le client sont répétées. Une solution plus efficace consiste à exécuter la requête pour qu'elle ajoute à chaque client la liste de ses commandes. La clause `Group Join` permet de réaliser ce regroupement. Il est également nécessaire dans ce cas de spécifier avec la clause`Into`, le nom de la propriété utilisée pour accéder au regroupement, dans notre cas la liste des commandes du client. À noter pour nous que cette propriété fera doublon avec celle que nous avons déjà prévue dans notre classe `Client`.

```
Dim requeteGroupJoin = From unclient In listeClients _
Group Join uneCommande In listeCommandes On unclient Equals uneCommande.LeClient _
Into CommandesDuClient = Group _
Select unclient, CommandesDuClient
   For Each r In requeteGroupJoin
       Console.WriteLine(r.unclient.nom)
       For Each c In r.CommandesDuClient
           Console.WriteLine(vbTab & c.dateCommande)
       Next
   Next
```

Un regroupement peut également être réalisé sans pour autant faire de jointure entre deux sources de données. Nous pouvons, par exemple, rechercher pour chaque ville la liste des clients y résidant. Pour cela, la clause `Group By Into` est idéale. Il suffit simplement d'indiquer que nous souhaitons regrouper les clients avec comme clé de regroupement leur ville de résidence et indiquer le nom de la propriété qui sera générée pour contenir le regroupement. La requête génère alors, lors de son exécution, une liste d'instances de classes contenant deux propriétés :

- Le nom de la ville.

- La liste des clients par l'intermédiaire de la propriété indiquée dans la requête.

```
Dim requeteGroup = From unclient In listeClients _
                   Group unclient By unclient.adresseFacture.ville _
                   Into ClientsParVille = Group _
                   Select ville, ClientsParVille
        For Each r In requeteGroup
            Console.WriteLine(r.ville)
            For Each c In r.ClientsParVille
                Console.WriteLine(vbTab & c.nom)
            Next
        Next
```

g. Quantificateurs

Les quantificateurs sont utilisés pour contrôler si dans une liste au moins un élément remplit une condition ou si tous les éléments remplissent une condition. Ils fournissent le résultat du contrôle sous forme d'un booléen qui en général est utilisé dans une clause`Where`. Comme exemple, nous recherchons les clients dont toutes les commandes ont été passées en 2007.

```
Dim requeteQuantifier = From unclient In listeClients _
Where (Aggregate cmd In unclient.LesCommandes Into All
(cmd.dateCommande.Year = 2007)) _
        Select unclient
For Each unclient In requeteQuantifier
    Console.WriteLine(unclient.nom)
    For Each c In unclient.LesCommandes
        Console.WriteLine(vbTab & c.dateCommande)
    Next
Next
```

La deuxième version permet de rechercher les clients ayant passé au moins une commande en 2008.

```
Dim requeteQuantifier1 = From unclient In listeClients _
Where (Aggregate cmd In unclient.LesCommandes Into Any
(cmd.dateCommande.Year = 2008)) _
                            Select unclient
        For Each unclient In requeteQuantifier1
            Console.WriteLine(unclient.nom)
            For Each c In unclient.LesCommandes
                Console.WriteLine(vbTab & c.dateCommande)
            Next
        Next
```

h. Agrégations

Les opérations d'agrégation sont utilisées pour le calcul d'une valeur unique à partir de valeurs contenues dans une liste d'éléments. Les opérations les plus courantes sont :

- le calcul de moyenne ;
- la recherche d'un maximum ;
- la recherche d'un minimum ;
- le calcul d'un total.

L'exemple suivant applique ces quatre opérateurs sur les frais de port de toutes les commandes.

```
Dim moyennePort = Aggregate unCommande In listeCommandes Into
Average(unCommande.port)
        Console.WriteLine("moyenne des frais de port : " & moyennePort)
```

```
        Dim maxiPort = Aggregate unCommande In listeCommandes Into
Max(unCommande.port)
        Console.WriteLine("maximum des frais de port : " & maxiPort)
        Dim miniPort = Aggregate unCommande In listeCommandes Into
Min(unCommande.port)
        Console.WriteLine("minimum des frais de port : " & miniPort)
        Dim totalPort = Aggregate unCommande In listeCommandes Into
Sum(unCommande.port)
        Console.WriteLine("total des frais de port : " & totalPort)
```

Ce code est également un bon exemple de requête exécutée immédiatement, puisque pour obtenir le résultat, la liste doit obligatoirement être parcourue du premier au dernier élément.

Après avoir étudié la syntaxe du langage, nous allons voir maintenant comment l'utiliser en association avec une base de données.

LINQ vers SQL

Comme nous l'avons vu dans les paragraphes précédents le domaine de prédilection de LINQ est le monde des objets. Il sait parfaitement manipuler les listes, les objets et les propriétés de ces objets. Pourtant ces éléments présentent un grave handicap : ils disparaissent inexorablement dès la fin de l'application. La solution la plus couramment utilisée pour pallier ce problème consiste à confier à une base de données le soin d'assurer la persistance des informations après l'arrêt de l'application. Le langage SQL est le plus couramment utilisé pour le dialogue avec une base de données. Bien que les principaux mots clés soient identiques les syntaxes de ces deux langages ne sont pas compatibles. Les requêtes LINQ sont donc automatiquement transformées en leurs homologues SQL pour réaliser les traitements. Un autre problème doit également être pris en compte. Jusqu'à présent nous avons manipulé, par l'intermédiaire des requêtes LINQ, des objets et rien que des objets. Le concept objet étant parfaitement étranger à une base de données il faut trouver une solution pour que LINQ puisse accéder aux informations. La clé de l'énigme consiste tout simplement à créer des classes pour représenter dans l'application les données présentes dans la base de données. Cette technique est également appelée mappage objet relationnel. Ce doit être la première étape dans l'utilisation de LINQ avec une base de données. Nous allons donc regarder comment créer ces classes.

1. Le mappage objet relationnel

Il existe trois solutions pour générer les classes représentant les informations stockées dans la base de données.

- Créer les classes manuellement comme n'importe laquelle des classes de votre application en utilisant un éditeur de code. Cette solution est extrêmement fastidieuse et n'est en général utilisée que pour les modifications minimes de classes existantes.
- Utiliser l'outil en ligne de commande SQLMetal.
- Utiliser le concepteur Objet/Relationnel en mode graphique.

a. SQLMetal

Cet outil est disponible à partir d'une fenêtre de commande de l'environnement Visual Studio. Les options indiquées sur la ligne de commande permettent de configurer le fonctionnement. Les options disponibles concernent :

- La génération à partir d'une base de données du code source des classes et des attributs de mappage.
- La génération à partir d'une base de données d'un fichier intermédiaire de mappage (.dbml).
- La génération à partir d'un fichier de mappage des classes et des attributs de mappage.

Chaque option doit être précédée par un caractère '/' et suivie par le caractère ' :' et de la valeur de l'option si besoin.

Les options de connexion :

/server: <nom du serveur>

Indique le nom ou l'adresse IP du serveur de base de données.

/database: <nom de la base de données>

Indique le nom de la base de données à partir de laquelle la génération doit être effectuée.

/user: <nom de connexion>

Indique le compte utilisateur avec lequel la connexion vers la base de données sera ouverte. Si cette option n'est pas spécifiée, c'est l'authentification Windows qui sera utilisée.

/password: <mot de passe>

Indique le mot de passe associé au compte utilisé pour établir la connexion.

/conn: <chaîne de connexion>

Peut être utilisée à la place des quatre options précédentes pour fournir les informations concernant la connexion vers le serveur de base de données.

timeout: <secondes>

Indique la durée maximum pendant laquelle SqlMetal tente d'établir la connexion vers la base de données. Une valeur égale à zéro indique une durée illimitée.

Les options de sortie :

/dbml :<nom du fichier>

Génère un fichier de mappage.

/code :<nom du fichier>

Génère le code source des classes dans le fichier indiqué.

Les options de génération :

/language :< vb ou csharp>

Indique le langage dans lequel le code sera généré. Les deux options valides sont vb pour Visual Basic et csharp pour C#.

/namespace :<nom>

Indique l'espace de nom dans lequel les classes seront générées. Par défaut, il n'y a pas d'espace de nom.

/context :<nom>

Spécifie le nom du data context généré. Par défaut, ce nom est déduit du nom de la base de données.

/entitybase :<nom>

Spécifie la classe de base des classes générées. Par défaut, les classes générées n'ont pas de classe de base.

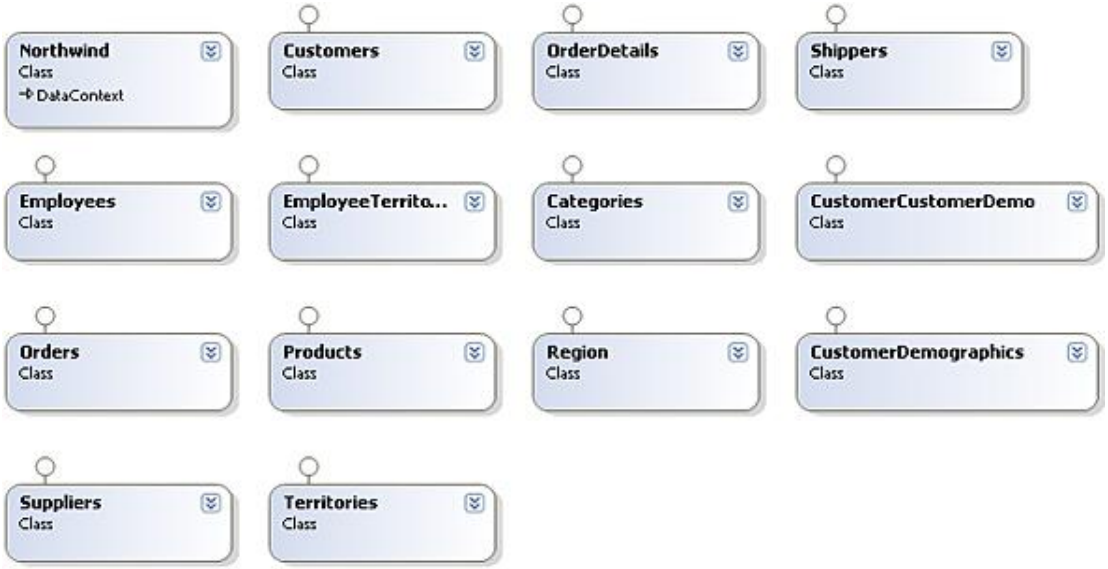
Enfin la dernière information à fournir correspond au nom du fichier de mappage à partir duquel la génération des classes sera réalisée. Cette information est inutile si la génération est exécutée directement depuis la base de données.

Voici quelques utilisations les plus courantes de cet outil.

Génération en Visual Basic des classes de la base Northwind située sur l'ordinateur local :

```
SqlMetal /server:localhost /database:northwind /language:vb /code:nw.vb
```

Le code généré étant trop volumineux pour le lister ici (environ 3500 lignes), voici simplement ci-après le diagramme des classes qui ont été générées.



Nous avons la classe Northwind qui hérite de la classe DataContext et qui va très rapidement nous servir pour que LINQ puisse dialoguer avec la base de données. Nous avons également une classe générée pour chacune des tables de la base de données. Ce sont des instances de ces classes que nous manipulerons dans l'application.

Génération du fichier de mappage de la base Northwind située sur l'ordinateur local :

```
SqlMetal /server:localhost /database:northwind /dbml:nw.dbml
```

Cette commande génère un fichier xml dont voici un extrait :

```
<Table Name="dbo.Customers" Member="Customers">
    <Type Name="Customers">
      <Column Name="CustomerID" Type="System.String" DbType="NChar(5) NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="CompanyName" Type="System.String" DbType="NVarChar(40)
NOT NULL" CanBeNull="false" />
      <Column Name="ContactName" Type="System.String" DbType="NVarChar(30)"
CanBeNull="true" />
      <Column Name="ContactTitle" Type="System.String" DbType="NVarChar(30)"
CanBeNull="true" />
      <Column Name="Address" Type="System.String" DbType="NVarChar(60)"
CanBeNull="true" />
      <Column Name="City" Type="System.String" DbType="NVarChar(15)"
CanBeNull="true" />
      <Column Name="Region" Type="System.String" DbType="NVarChar(15)"
CanBeNull="true" />
      <Column Name="PostalCode" Type="System.String" DbType="NVarChar(10)"
CanBeNull="true" />
      <Column Name="Country" Type="System.String" DbType="NVarChar(15)"
CanBeNull="true" />
      <Column Name="Phone" Type="System.String" DbType="NVarChar(24)"
CanBeNull="true" />
      <Column Name="Fax" Type="System.String" DbType="NVarChar(24)"
CanBeNull="true" />
      <Association Name="FK_CustomerCustomerDemo_Customers" Member=
"CustomerCustomerDemo" OtherKey="CustomerID" Type="CustomerCustomerDemo"
DeleteRule="NO ACTION" />
      <Association Name="FK_Orders_Customers" Member="Orders" OtherKey=
"CustomerID" Type="Orders" DeleteRule="NO ACTION" />
    </Type>
</Table>
```

Ce fichier peut être modifié pour, par exemple, changer le nom des classes et des propriétés associées aux informations en provenance de la base de données. Dans l'exemple ci-dessous, nous avons francisé les noms.

```
<Table Name="dbo.Customers" Member="Clients">
    <Type Name="Clients">
      <Column Name="CustomerID" Member="CodeClient" Storage="_CustomerID"
```

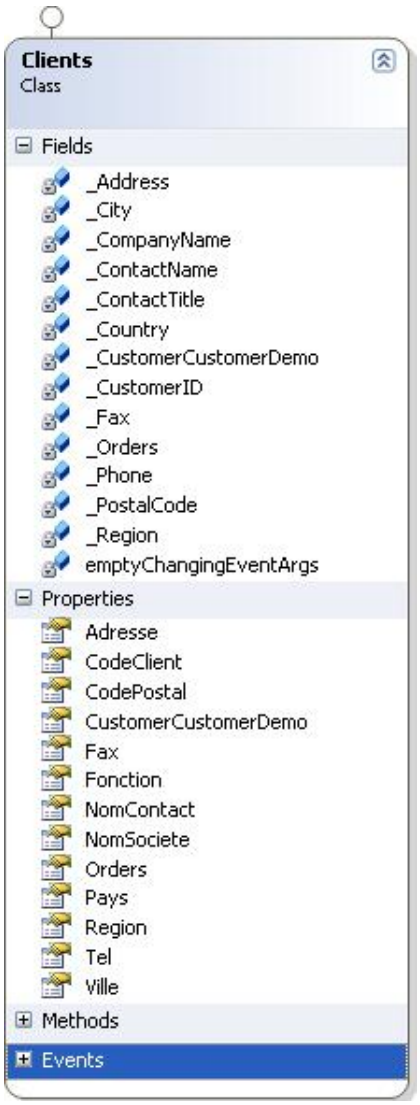
```
Type="System.String" DbType="NChar(5) NOT NULL" IsPrimaryKey="true"
CanBeNull="false" />
      <Column Name="CompanyName" Member="NomSociete" Storage="_CompanyName"
Type="System.String" DbType="NVarChar(40) NOT NULL" CanBeNull="false" />
      <Column Name="ContactName" Member="NomContact" Storage="_ContactName"
Type="System.String" DbType="NVarChar(30)" CanBeNull="true" />
      <Column Name="ContactTitle" Member="Fonction" Storage="_ContactTitle"
Type="System.String" DbType="NVarChar(30)" CanBeNull="true" />
      <Column Name="Address" Member="Adresse" Storage="_Address"
Type="System.String" DbType="NVarChar(60)" CanBeNull="true" />
      <Column Name="City" Member="Ville" Storage="_City" Type="System.String"
DbType="NVarChar(15)" CanBeNull="true" />
      <Column Name="Region" Type="System.String" DbType="NVarChar(15)"
CanBeNull="true" />
      <Column Name="PostalCode" Member="CodePostal" Storage="_PostalCode"
Type="System.String" DbType="NVarChar(10)" CanBeNull="true" />
      <Column Name="Country" Member="Pays" Storage="_Country"
Type="System.String" DbType="NVarChar(15)" CanBeNull="true" />
      <Column Name="Phone" Member="Tel" Storage="_Phone" Type="System.String"
DbType="NVarChar(24)" CanBeNull="true" />
      <Column Name="Fax" Type="System.String" DbType="NVarChar(24)"
CanBeNull="true" />
      <Association Name="Clients_CustomerCustomerDemo" Member=
"CustomerCustomerDemo" ThisKey="CodeClient" OtherKey="CustomerID"
       Type="CustomerCustomerDemo" />
      <Association Name="Clients_Orders" Member="Orders" ThisKey="CodeClient"
OtherKey="CustomerID" Type="Orders" />
    </Type>
  </Table>
```

Dans cet exemple, pour pouvoir utiliser des noms de propriétés différents des noms de colonnes dans la base de données, nous avons ajouté l'attribut `Member` à chaque balise <Column> pour spécifier le nom de la propriété et l'attribut `Storage` pour indiquer le nom de la variable interne de la classe qui contiendra l'information.

Nous pouvons maintenant générer le code à partir du fichier de mappage modifié avec la commande suivante.

```
SqlMetal /code:nw.vb /language:vb nw.dbml
```

Il nous reste à visualiser la classe générée pour vérifier que nos modifications ont bien été prises en compte.



Cet outil est très facile à utiliser mais présente le petit inconvénient de ne pouvoir générer les classes que pour l'intégralité d'une base de données. De plus, les éventuelles modifications sont à faire manuellement soit dans le code source généré soit dans le fichier de mappage intermédiaire. Pour la génération et la personnalisation de quelques classes, il est préférable d'utiliser le concepteur Objet/Relationnel intégré à Visual Studio.

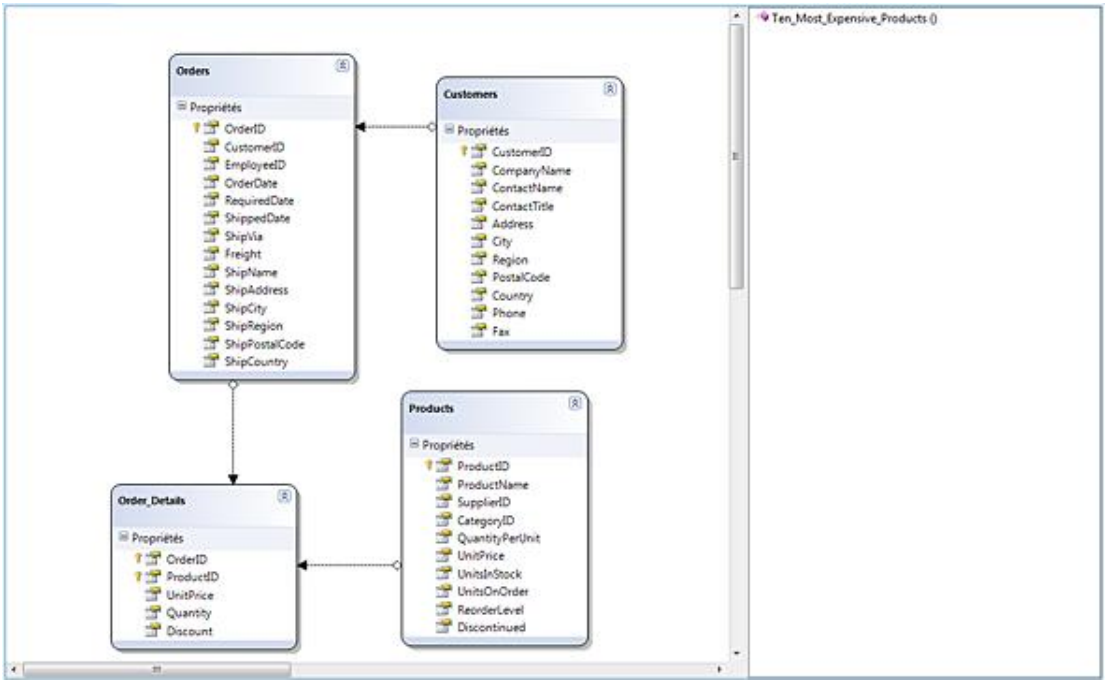
b. Concepteur Objet/Relationnel

Le concepteur Objet/Relationnel fournit une solution très pratique pour créer le modèle objet d'une application représentant les informations disponibles dans une base de données. Il permet également la création de procédures et fonctions autorisant l'utilisation des procédures stockées et fonctions présentes dans la base de données. Il comporte cependant quelques limitations :

- seules les bases de données SQL Server 2000, SQL Server 2005, SQL Server Express sont supportées.
- Le mappage n'est possible qu'entre une classe et une table. C'est-à-dire qu'il n'est pas possible de créer une classe pour représenter le résultat d'une jointure entre plusieurs tables.
- Le concepteur fonctionne en 'sens unique' car seules les modifications effectuées dans le concepteur sont répercutées dans le code généré. Si le code est modifié manuellement les modifications ne sont pas prises en compte dans le concepteur. Pire encore si des modifications sont faites dans le concepteur après des modifications manuelles du code, ces modifications sont perdues lors de l'enregistrement du concepteur car dans ce cas le code est régénéré automatiquement. La solution consiste à créer une classe partielle dans un fichier indépendant de celui manipulé par le concepteur.

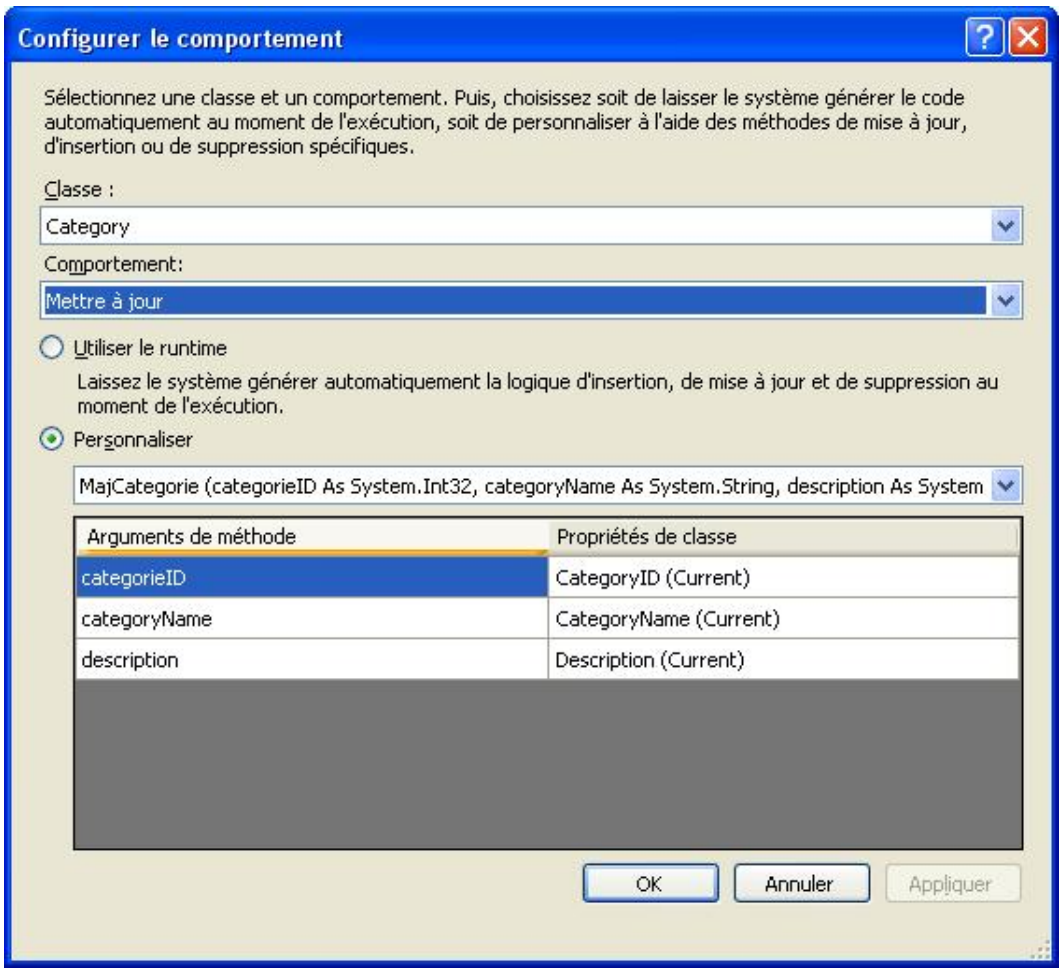
Le concepteur Objet/Relationnel est lancé automatiquement lors de l'ajout d'un élément de type LINQ to SQL Classes. L'ajout à un projet d'un fichier .dbml provoque également l'ouverture de cet outil. À l'ouverture, la surface du concepteur est séparée en deux parties. La zone de gauche va accueillir les classes associées aux tables alors que la

zone de droite va accueillir les procédures et fonctions associées aux procédures stockées. L'ensemble représente le DataContext généré.



Ajout de classes

Vous pouvez créer les classes représentant les tables d'une base de données en réalisant un glisser-déplacer d'une ou de plusieurs tables à partir de l'explorateur de serveurs vers la partie gauche du concepteur. Le premier élément ajouté au concepteur Objet/Relationnel est également utilisé pour configurer les propriétés de connexion du DataContext. Si un autre élément provenant d'une autre base de données est ajouté, une boîte de dialogue vous demande alors si vous souhaitez remplacer la connexion existante. Si vous acceptez la modification les classes déjà présentes dans le concepteur ne pourront bien sur plus être utilisées. L'ajout d'une table génère le code nécessaire pour que le DataContext initialise les propriétés d'une instance de la classe à partir des informations présentes dans une ligne de la base de données. Il ajoute également le code nécessaire pour que les modifications apportées aux propriétés de l'instance puissent être répercutées dans la base de données. Le concepteur se base sur la structure de la table et sur la clé primaire pour effectuer les mises à jour. Vous pouvez également indiquer vous-même comment seront effectuées les mises à jour. Pour cela, chaque classe possède trois propriétés Insert, Update, Delete. Par défaut, ces propriétés sont initialisées avec la valeur Use Runtime pour indiquer que le code chargé des mises à jour est généré automatiquement. Pour modifier ce comportement vous pouvez assigner à ces propriétés des procédures stockées. Une boîte de dialogue permet la configuration de ces propriétés.

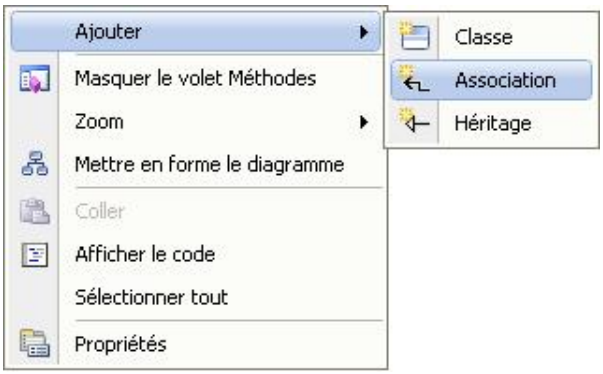


Après la sélection de la procédure stockée vous devez indiquer comment seront renseignés les paramètres attendus en entrée par la procédure stockée. Les valeurs disponibles correspondent aux différentes propriétés de la classe. Pour chaque propriété la version actuelle ou originale est disponible.

Ajout d'associations

Après avoir déposé plusieurs tables sur le concepteur il est ensuite possible de créer des associations entre certaines d'entre elles. Les associations sont tout à fait similaires aux relations entre tables dans une base de données. D'ailleurs s'il existe dans la base de données une relation de clé étrangère entre deux tables une association sera créée automatiquement lorsque ces deux tables sont déposées sur le concepteur.

Pour ajouter manuellement une association vous devez passer par le menu contextuel du concepteur Objet/Relationnel.



Une boîte de dialogue vous propose alors de configurer la relation. Vous devez donc choisir la classe parente et la classe enfant de la relation. La classe parente est la classe qui se trouve à l'extrémité 'un' d'une relation un à plusieurs, la classe enfant représente l'extrémité 'plusieurs' de la relation. Par exemple, dans l'association entre les classes Product et Category, la classe Category représente le côté 'un' de la relation et la classe Product le côté 'plusieurs'. En effet un produit appartient à une catégorie et une catégorie contient plusieurs produits. Vous devez ensuite indiquer pour chacune des classes la ou les propriétés qui vont participer à la relation. Il faut veiller à ce que les propriétés participant à l'association soit du même type à chaque extrémité de l'association. Après la création de

l'association, vous pouvez configurer certaines propriétés qui ne sont pas disponibles au moment de la création.

- `Cardinality` : détermine si l'association est de type un à plusieurs (one-to-many) ou de type un à un (one-to-one).

- `Child Property` : indique si une propriété doit être créée dans la classe parente pour référencer les informations de la classe enfant. Le type de cette propriété est déterminé par le type de la classe enfant et la cardinalité. Si la cardinalité est un à un, la propriété est une simple référence vers une instance de la classe concernée. Si la cardinalité est un à plusieurs la propriété est une collection d'instances de la classe concernée.

- Les propriétés `Name` permettent d'identifier les propriétés créées pour réaliser l'association.

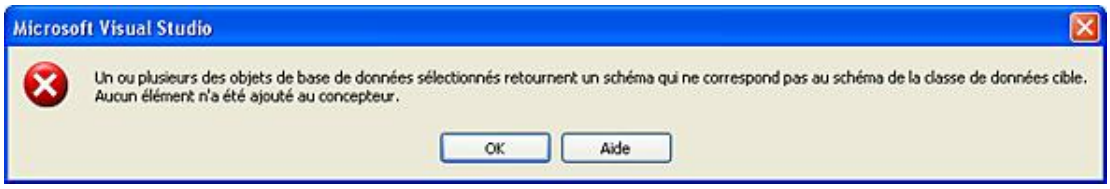
Ajout de méthodes

Les procédures stockées et les fonctions peuvent être ajoutées au concepteur Objet/Relationnel pour ensuite être transformées en méthodes du DataContext. L'appel de ces méthodes provoquera l'exécution de la procédure stockée ou de la fonction par le serveur de base de données. Si des paramètres sont attendus en entrée par la procédure stockée, ils devront être fournis à la méthode lors de son exécution. L'ajout d'une méthode au DataContext se réalise très simplement en effectuant un glisser-déplacer entre l'explorateur de serveurs et le concepteur Objet/Relationnel. Il faut cependant être attentif lorsque vous ajoutez une procédure stockée ou une fonction car l'emplacement où aura lieu le déplacement détermine le type de retour de la méthode générée. Si l'élément est déplacé vers la zone de droite du concepteur, le type de retour sera généré automatiquement.

Par contre, si l'élément est déplacé vers une classe existante du concepteur, le type de retour correspondra à cette classe à condition toutefois que l'information renvoyée par la procédure stockée soit compatible avec cette classe. Nous allons faire quelques manipulations avec la procédure stockée `[Ten Most Expensive Products]` dont voici le code :

```
set ANSI_NULLS ON
set QUOTED_IDENTIFIER ON
go

CREATE procedure [dbo].[Ten Most Expensive Products] AS
SET ROWCOUNT 10
SELECT Products.ProductName AS TenMostExpensiveProducts, Products.UnitPrice
FROM Products
ORDER BY Products.UnitPrice DESC
```

Cette procédure retourne le nom et le prix des dix produits les plus onéreux. Si nous essayons d'ajouter cette procédure au DataContext en effectuant un glisser-déplacer sur la surface de la classe `Product` nous obtenons le message suivant.

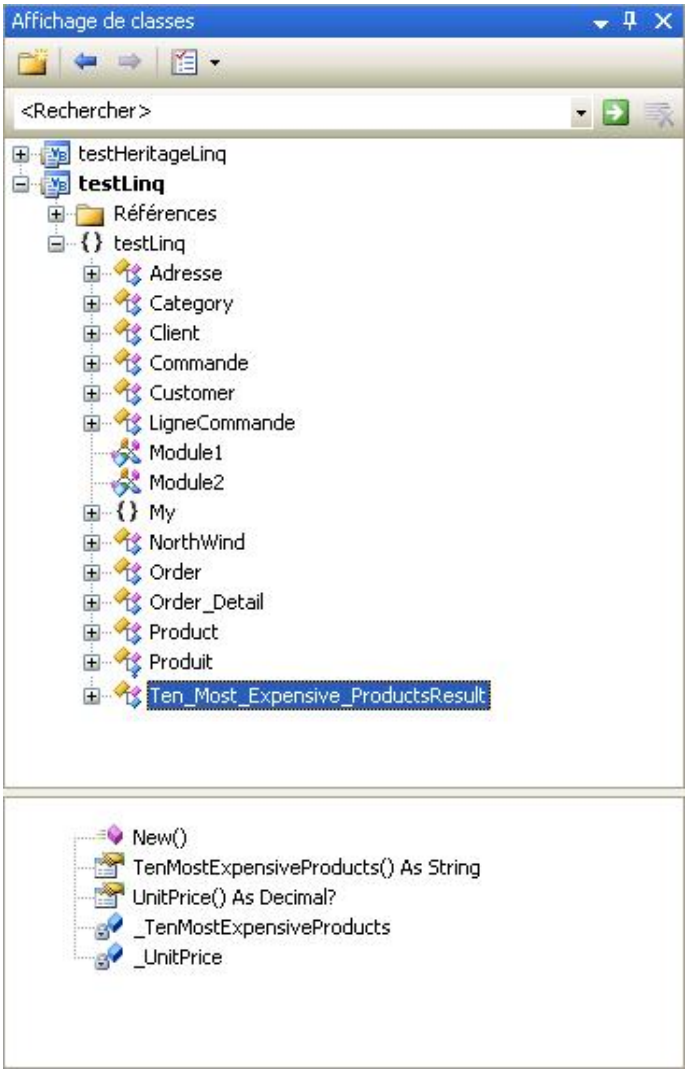


En effet les éléments renvoyés par la procédure stockée ne sont pas des produits mais simplement le nom et le prix du produit.

Par contre, si nous réalisons un glisser déplacer vers la zone de droite du concepteur, l'opération se réalise sans problème. La fonction suivante est ajoutée au DataContext.

```
Public Function Ten_Most_Expensive_Products() As ISingleResult
(Of Ten_Most_Expensive_ProductsResult)

   Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.
GetCurrentMethod,MethodInfo))

   Return CType(result.ReturnValue,ISingleResult(Of Ten_Most_Expensive_
ProductsResult))
End Function
```

Cette fonction ne retourne pas une liste de produits mais une liste de `Ten_Most_Expensive_ProductsResult` qui correspond à une classe générée automatiquement en fonction des informations renvoyées par la procédure stockée dont voici la structure.



Pour que la classe `Product` puisse être utilisée comme type de retour pour la fonction, nous sommes obligés de légèrement modifier la procédure stockée pour qu'elle retourne des produits et non juste le nom et le prix du produit.

```
set ANSI_NULLS ON
set QUOTED_IDENTIFIER ON
go


ALTER procedure [dbo].[Ten Most Expensive Products] AS
SET ROWCOUNT 10
SELECT *
FROM Products
ORDER BY Products.UnitPrice DESC
```

Nous pouvons maintenant refaire la même opération et obtenir la fonction suivante qui cette fois, retourne bien une liste de produits.

```
Public Function Ten_Most_Expensive_Products() As ISingleResult(Of Product)
      Dim result As IExecuteResult = Me.ExecuteMethodCall(Me, CType(MethodInfo.
GetCurrentMethod,MethodInfo))
      Return CType(result.ReturnValue,ISingleResult(Of Product))
End Function
```

Héritage de classes

Comme n'importe quelle classe, les classes générées par le Concepteur Objet/Relationnel peuvent utiliser l'héritage. Par contre, côté base de données, c'est une notion parfaitement inconnue. Il faut avoir recours à quelques petites astuces pour simuler cette technique dans une base de données. La solution fréquemment utilisée, consiste à créer une table unique qui contiendra à la fois les informations des objets de la classe de base et les informations de la

sous-classe. Une colonne supplémentaire est ajoutée à la table et sert de discriminateur. En fonction de la valeur de cette colonne, il est facile de déterminer si la ligne doit être représentée par une instance de la classe de base ou une instance de la sous-classe. Nous allons modifier la table Products pour pouvoir mettre cette technique en application. Nous ajoutons à la table une colonne nommée Périssable de type entier. Cette colonne va nous servir de discriminateur. Si la valeur qu'elle contient est égale à 1, il s'agit d'un produit non périssable. Si la valeur est égale à 2, il s'agit d'un produit périssable. Dans ce cas la deuxième colonne ajoutée, nommée DLC de type date, contient la date limite de consommation du produit. Pour les produits non périssable cette colonne ne contient aucune valeur (null). Modifiez les informations pour quelques produits afin qu'ils deviennent des produits périssables (Camembert Pierrot, Escargots de Bourgogne, Mascarpone Fabioli…).

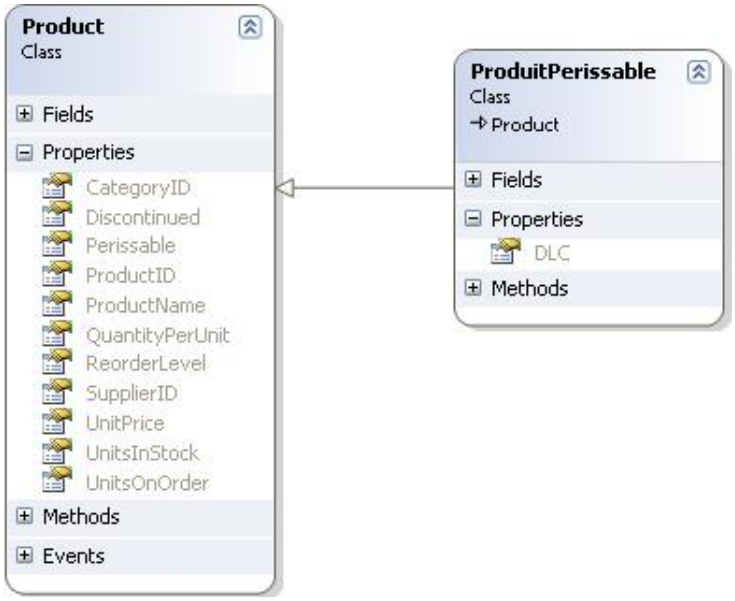
Maintenant que la base de données est prête, nous pouvons ajouter les classes au DataContext. La première étape consiste à ajouter la table constituant la classe de base. Puis ajoutez un deuxième exemplaire de cette table et renommez la classe correspondante qui va devenir la classe dérivée. Ajoutez ensuite à partir de la boîte à outils une relation d'héritage entre les deux classes en la dessinant depuis la classe fille vers la classe parente. Dans chacune des classes, supprimez les propriétés inutiles. Par exemple, en ne conservant dans la classe dérivée que la propriété DLC et en la supprimant dans la classe de base. Après avoir sélectionné la relation d'héritage sur le diagramme vous devez modifier ses propriétés.

- indiquez par l'intermédiaire de la propriété`Discriminator Property`, la propriété servant à faire la distinction entre une instance de la classe de base et une instance de la sous classe.

- Configurez ensuite les propriétés `Base Class Discriminator Value` et `Derived Class Discriminator Value`avec les valeurs de la propriété configurée précédemment représentant une instance de la classe de base et une instance de la sous classe.

- La propriété `Inheritance Default`indique quelle classe sera utilisée si le discriminateur contient une valeur inconnue.

Nous obtenons les classes suivantes :



Maintenant que nos classes sont disponibles, regardons comment les utiliser par l'intermédiaire de requêtes LINQ.

c. Utilisation de requêtes LINQ vers SQL

Les requêtes LINQ pour SQL utilisent rigoureusement la même syntaxe que celles que nous avons étudiées dans le paragraphe 2. La seule petite distinction provient des données qui dans ce cas, sont extraites de la base de données et transformées en instances de classes à partir des informations de mappage. Le dialogue avec la base de données est entièrement pris en charge par le DataContext. Il faut donc créer une instance du DataContext et c'est par son intermédiaire que les données seront disponibles pour l'exécution de la requête LINQ. Voici ci-dessous notre première requête LINQ vers la base de données.

```
Dim dc As NorthWind
        dc = New NorthWind
        Dim requete = From unClient In dc.Customers _
               Where unClient.ContactName Like "A*" _
               Select unClient
```

```
        For Each Client In requete
            Console.WriteLine(Client.ContactName)
        Next
```

Dans ce code, la classe NorthWind correspond au DataContext et c'est donc par son intermédiaire que les données sont disponibles pour la requête LINQ. Mais comment les données sont-elles sélectionnées ?

En fait la méthode la plus naturelle pour obtenir des informations en provenance d'une base de données est de demander à celle-ci d'exécuter une requête SQL. C'est effectivement cette solution qui est utilisée par LINQ. Pour le vérifier, nous pouvons demander au DataContext (NorthWind dans notre cas) d'afficher sur la console le code SQL qu'il génère automatiquement. Pour cela, il suffit simplement d'initialiser la propriété Log du DataContext en direction de la console avant la création de la requête LINQ.

```
Dim dc As NorthWind
        dc = New NorthWind
        dc.Log = Console.Out
        Dim requete = From unClient In dc.Customers _
               Where unClient.ContactName Like "A*" _
               Select unClient

        For Each Client In requete
            Console.WriteLine(Client.ContactName)
        Next
```

À l'exécution, nous obtenons l'affichage suivant :

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[ContactName] LIKE @p0
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [A%]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8

Ana Trujillo
Antonio Moreno
Ann Devon
Aria Cruz
André Fonseca
Annette Roulet
Alexander Feuer
Alejandra Camino
Art Braunschweiger
Anabela Domingues
```

Nous avons effectivement une requête SQL avec paramètres qui est créée automatiquement par le DataContext. Cette requête n'est pas très compliquée et elle aurait été facilement écrite en utilisant directement ADO.NET. Essayons d'exécuter une autre requête en reprenant la requête nous permettant d'obtenir les dates de commande de chacun des clients.

```
Dim requeteGroupJoin = From unclient In dc.Customers _
Group Join uneCommande In dc.Orders On unclient Equals uneCommande.Customer _
Into CommandesDuClient = Group _
Select unclient, CommandesDuClient
        For Each r In requeteGroupJoin
            Console.WriteLine(r.unclient.ContactName)
            For Each c In r.CommandesDuClient
                Console.WriteLine(vbTab & c.OrderDate)
            Next
        Next
```

Voici le code SQL généré pour l'exécution de cette requête :

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName], [t0].[Contact
Title], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Coun
try], [t0].[Phone], [t0].[Fax], [t1].[OrderID], [t1].[CustomerID] AS [CustomerID
2], [t1].[EmployeeID], [t1].[OrderDate], [t1].[RequiredDate], [t1].[ShippedDate]
, [t1].[ShipVia], [t1].[Freight], [t1].[ShipName], [t1].[ShipAddress], [t1].[Ship
City], [t1].[ShipRegion], [t1].[ShipPostalCode], [t1].[ShipCountry], (
```

```
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
    ) AS [value]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

Cela commence à sérieusement se compliquer. L'écriture d'une telle requête directement en SQL demanderait certainement une bonne maîtrise de ce langage alors que la syntaxe LINQ demeure très simple. C'est effectivement à ce niveau que réside la puissance de LINQ vers SQL.

Cette facilité ne se limite pas à l'extraction d'informations depuis la base de données car LINQ vers SQL est également capable de gérer les mises à jour des informations vers la base de données.

d. Mise à jour des données

La mise à jour de la base de données se réalise également très simplement uniquement en manipulant des objets et sans écrire la moindre ligne de SQL.

Modification de données existantes

La première étape consiste à obtenir les données que l'on souhaite modifier en exécutant une requête de sélection ordinaire. Une fois que les données sont disponibles sous forme d'instances de classes, nous pouvons simplement modifier les propriétés de ces instances. Pour transférer les modifications dans la base de données, il suffit simplement de demander au DataContext de propager les modifications vers la base de données. Nous allons tester cette technique en faisant déménager nos clients Nantais vers Saint Herblain.

```
Dim clientsNantais = From unclient In dc.Customers _
                     Where unclient.City = "Nantes" _
                     Select unclient

For Each unclient In clientsNantais
    unclient.City = "Saint Herblain"
    unclient.PostalCode = "44800"
Next
dc.SubmitChanges()
```

Dans ce code, c'est l'instruction `SubmitChanges`du DataContext qui provoque la mise à jour de la base de données en exécutant automatiquement la requête SQL Update suivante pour chaque objet ayant été modifié.

```
UPDATE [dbo].[Customers]
SET [City] = @p10, [PostalCode] = @p11
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2)
AND ([ContactTitle] = @p3) AND ([Address] = @p4) AND ([City] = @p5)
AND ([Region]IS NULL) AND ([PostalCode] = @p6) AND ([Country] = @p7)
AND ([Phone] = @p8) AND ([Fax] = @p9)
-- @p0: Input NChar (Size = 5; Prec = 0; Scale = 0) [FRANR]
-- @p1: Input NVarChar (Size = 19; Prec = 0; Scale = 0) [France restauration]
-- @p2: Input NVarChar (Size = 14; Prec = 0; Scale = 0) [Carine Schmitt]
-- @p3: Input NVarChar (Size = 17; Prec = 0; Scale = 0) [Marketing Manager]
-- @p4: Input NVarChar (Size = 14; Prec = 0; Scale = 0) [54, rue Royale]
-- @p5: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [Nantes]
-- @p6: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [44000]
-- @p7: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [France]
-- @p8: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [40.32.21.21]
-- @p9: Input NVarChar (Size = 11; Prec = 0; Scale = 0) [40.32.21.20]
-- @p10: Input NVarChar (Size = 14; Prec = 0; Scale = 0) [Saint Herblain]
-- @p11: Input NVarChar (Size = 5; Prec = 0; Scale = 0) [44800]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Suppression de données

Comme pour la modification, nous devons au préalable obtenir les éléments que nous souhaitons supprimer en exécutant une requête de sélection, puis indiquer pour chacun d'entre eux que nous souhaitons les supprimer. Pour cela, nous appelons la méthode `DeleteOnSubmit` de la table à laquelle appartient l'élément, en lui passant comme paramètre l'objet à supprimer. Pour valider les suppressions, nous devons ensuite appeler la méthode `SubmitChanges` du DataContext. Nous allons tester cela en supprimant les clients Brésiliens de la base de données.

```
Dim suppressionClient = From unclient In dc.Customers _
                        Where unclient.Country = "Brazil" _
                        Select unclient
        For Each unclient In suppressionClient
            dc.Customers.DeleteOnSubmit(unclient)
        Next
        dc.SubmitChanges()
```

À l'exécution de ce code, nous obtenons cette magnifique Exception.

Dans notre précipitation, nous avons simplement oublié un petit détail. Dans la base de données, les tables Customers, Orders et OrdersDetails sont liées par des contraintes de clé étrangère. Il est donc impossible de supprimer un client s'il possède encore des commandes et de la même façon il est impossible de supprimer une commande si elle contient encore des lignes de commande. Le problème vient du fait que LINQ n'est pas capable de gérer les suppressions en cascade. Pour résoudre notre problème nous avons deux solutions :

- Activer la règle ON DELETE CASCADE sur les contraintes de clé étrangères.
- Gérer nous même la suppression des objets enfants avant la suppression des objets parents.

C'est cette dernière solution que nous allons utiliser. Puisque notre modèle objet est correctement conçu, cette solution est très facile à mettre en œuvre. En effet dans la classe Customers, nous avons la collection Orders qui représente les commandes du client. De même dans la classe Orders, nous avons la collection Order_Details qui représente toutes les lignes d'une commande. Il suffit simplement d'exécuter trois boucles imbriquées qui vont supprimer les lignes de chaque commande, les commandes de chaque client puis les clients eux-mêmes.

```
Dim suppressionClient = From unclient In dc.Customers _
                        Where unclient.Country = "Brazil" _
                        Select unclient
        For Each unclient In suppressionClient
            For Each uneCommande In unclient.Orders
                For Each uneLigne In uneCommande.Order_Details
                    dc.Order_Details.DeleteOnSubmit(uneLigne)
                Next
                dc.Orders.DeleteOnSubmit(uneCommande)
            Next
            dc.Customers.DeleteOnSubmit(unclient)
        Next
        dc.SubmitChanges()
```

Avec cette solution, il n'y a plus de problèmes et nos clients Brésiliens sont bien effacés de la base de données.

Ajout de données

L'ajout de données se réalise en trois étapes. Il faut tout d'abord créer une instance de la classe représentant les données à insérer dans la base. Les propriétés de cette instance sont ensuite initialisées avec les valeurs que l'on souhaite ajouter dans la base de données. L'objet ainsi configuré doit être ensuite inséré dans la table du DataContext. Finalement les modifications sont transférées vers la base de données. Pour illustrer ces étapes, nous allons ajouter un nouveau client dans la base de données.

```
Dim nouveauClient As Customer
```

```
nouveauClient = New Customer
With nouveauClient
    .CustomerID = "MDUPO"
    .ContactName = "Michel Dupond"
    .CompanyName = "ENI"
    .ContactTitle = "Formateur"
    .Country = "France"
    .City = "Saint Herblain"
    .Address = "rue Benjamin Franklin"
    .Fax = "02.28.03.17.29"
    .Phone = "02.28.03.17.28"
    .PostalCode = "44800"
End With
dc.Customers.InsertOnSubmit(nouveauClient)
dc.SubmitChanges()
```

e. Conflits des mises à jour

Il arrive fréquemment que plusieurs utilisateurs travaillent simultanément sur la même base de données. Il peut se produire des conflits lorsque les mêmes enregistrements de la base de données sont mis à jour par plusieurs utilisateurs. LINQ propose un mécanisme permettant de traiter ce problème. Ce mécanisme se décompose en quatre étapes :

- configurer pour quelles informations de la base de données les conflits seront surveillés,
- détecter qu'un conflit survient,
- obtenir des informations sur le conflit,
- résoudre le conflit.

Configuration des classes pour la détection des conflits

Lors de la création des classes avec le concepteur Objet/Relationnel nous pouvons indiquer pour chaque propriété si elle doit être incluse dans le mécanisme de détection des conflit. Chaque membre de la classe générée possède une propriété `Update Check` à laquelle nous pouvons affecter trois valeurs différentes :

- Always : la détection des conflits est toujours active pour cet élément.
- WhenChanged : active la détection uniquement si la valeur a été modifiée.
- Never : ne pas tenir compte de cet élément pour la détection des conflits.

Par défaut, toutes les propriétés sont utilisées pour la détection des conflits.

Détection des conflits

Les conflits surviennent lors du transfert des informations vers la base de données. C'est donc à ce niveau que nous devons intervenir. Pour cela, lors de l'appel de la méthode `SubmitChanges` du DataContext, nous indiquons en passant un paramètre à cette mé- thode comment doit se comporter le mécanisme de détection des conflits. Deux solutions sont possibles :

- FailOnFirstConflict : signale le problème dès que le premier conflit intervient.
- ContinueOnConflict : essaie d'effectuer toutes les mises à jour et signale à la fin si un conflit est survenu.

Si un conflit est détecté une exception de type`ChangeConflictException` est déclenchée. L'appel de la méthode `SubmitChanges` doit donc être placée dans un bloc Try Catch pour récupérer l'exception.

Obtenir les informations sur les conflits

Dans le bloc Catch, nous pouvons obtenir des informations sur les conflits en parcourant la collection`ChangeConflicts`

du DataContext. Cette collection contient un ou plusieurs objets de typeObjectChangeConflict. La propriété Object nous permet d'obtenir une référence sur l'élément à l'origine du problème. La propriété MemberConflicts fournit quant à elle, la liste de tous les membres de cet objet qui sont à l'origine du problème.

Pour chacun, nous avons à notre disposition la valeur originale au moment de la création de l'instance de la classe à partir des informations de la base de données, la valeur actuelle pour l'instance de la classe, la valeur actuelle dans la base de données.

Résoudre les conflits

Pour résoudre les conflits survenus lors de la mise à jour de la base de données trois hypothèses sont envisageables.

- Remplacer les valeurs des propriétés en conflits avec les informations présentes dans la base de données. Il faut pour cela appeler la méthode Resolve de l'objet ObjectChangeConflict en lui passant la constante Overwrite CurrentValues.

- Remplacer les valeurs de la base de données par les informations contenues dans les propriétés de l'objet. Comme pour la solution précédente, nous devons appeler la méthode Resolve de l'objet ObjectChangeConflict en lui passant cette fois la constante KeepCurrentValues.

- Fusionner les propriétés de l'objet avec les informations de la base de données. Les informations de la base de données ne sont modifiées que si la propriété correspondante de l'objet a été modifiée. La méthode Resolve doit dans ce cas être appelée avec la constante KeepChanges.

Le code suivant vous permet de tester ces différentes solutions en remplaçant simplement la constante lors de l'appel de la méthode Resolve.

```
Dim cl As Customer
Dim rqt = From unClient In dc.Customers _
          Where unClient.CustomerID = "BOLID" _
          Select unClient

For Each cl In rqt
    cl.City = "Barcelone"
Next
Console.WriteLine("modifier le code postal du client BOLID dans la base puis
taper une touche")
Console.ReadLine()
Try
   dc.SubmitChanges(ConflictMode.FailOnFirstConflict)
Catch ex As ChangeConflictException
      For Each o As ObjectChangeConflict In dc.ChangeConflicts
          o.Resolve(RefreshMode.KeepChanges)
        ' o.Resolve(RefreshMode.KeepCurrentValues)
        ' o.Resolve(RefreshMode.OverwriteCurrentValues)
      Next
End Try
Console.WriteLine("l'objet client :")
Console.WriteLine("city:" & cl.City)
Console.WriteLine("postalCode:" & cl.PostalCode)
For Each cli In rqt
    Console.WriteLine("le client dans la base :")
    Console.WriteLine("city:" & cli.City)
    Console.WriteLine("postalCode:" & cli.PostalCode)
Next
```

Présentation

Le langage XML (*eXtensible Markup Language*) est un langage permettant la représentation de données. Il permet d'encapsuler tout type de données, en les représentant sous la forme d'une arborescence. Celles-ci sont écrites entre des balises ou sous forme d'attributs. Ce format permet de décrire des données mais ne permet pas de mettre en forme ni de les exploiter. Il est principalement utilisé pour permettre l'échange de données entre applications et même entre systèmes différents. Il est, également, souvent utilisé comme format de stockage pour les paramètres de configuration d'une application. Visual Studio et Windows l'utilisent à cet effet de manière courante. Ce langage a été conçu par le W3C (*World Wide Web Consortium*). C'est donc sur le site **http://www.w3.org/XML** que vous pourrez obtenir le détail des spécifications de ce langage.

Le langage XML est souvent confondu avec le langage HTML. Bien que comportant des similitudes, ces deux langages n'ont pas la même vocation. Voici les points communs entre les langages XML et HTML :

- Ces deux langages se présentent sous forme "texte seulement".
- Le contenu des documents est représenté au moyen de balises.
- Ces balises peuvent comporter des attributs.
- Les balises peuvent être imbriquées les unes à l'intérieur des autres.
- Ces deux langages sont issus tous deux d'une même base : le SGML (*Standard Generalized Markup Language*).

Le langage XML se distingue du langage HTML par les points suivants :

- Le langage XML autorise la création de vos propres balises.
- Les outils chargés du traitement gèrent la syntaxe de façon plus rigoureuse.
- HTML est un langage conçu pour la présentation des données. À l'inverse, XML est utilisé pour la description des données.

Pour pouvoir être facilement manipulées, les données XML doivent être confiées à un processeur XML.

Un processeur XML est un module logiciel, spécialement écrit pour manipuler XML. Le recours à un module externe pour le traitement XML s'explique par la complexité que représente le développement d'un processeur XML, totalement fonctionnel. En effet, pour qu'un processeur XML soit considéré totalement fonctionnel, son fonctionnement doit suivre les évolutions du langage définies par le W3C. Il est donc important de visiter régulièrement le site Microsoft pour vérifier s'il existe une version plus récente du processeur XML que celle installée sur votre machine.

Structure d'un document XML

Avant de manipuler des documents XML à partir de Visual Basic, il est important de bien comprendre la structure de ce type de document. Les paragraphes suivants vont présenter les notions élémentaires à connaître avant de se lancer dans l'utilisation de documents XML.

1. Constituants d'un document XML

Un document XML peut être constitué des éléments suivants :

Instruction de traitement

Les instructions de traitement permettent d'incorporer, dans un document XML, des informations destinées au processeur XML ou à d'autres applications devant manipuler le document. Ces instructions de traitement sont utilisées pour fournir une instruction spéciale à une application travaillant sur le document.

L'instruction de traitement est insérée dans le document avec la syntaxe suivante :

```
< ?nomApplication instruction ?>
```

La première partie est le nom de l'application à qui est destinée cette instruction. La deuxième partie est le texte de l'instruction.

Un document XML contient en général une instruction de traitement spéciale pour définir la version de XML avec laquelle le document est conforme et le codage des caractères utilisé par le document.

```
<?xml version="1.0" encoding="utf-8" ?>
```

Commentaires

Les commentaires servent à inclure dans le document des informations destinées aux utilisateurs du document. Ils sont ignorés par le processeur XML ou par les applications utilisant le document. Ils ne doivent pas être incorporés dans une balise.

La syntaxe suivante doit être utilisée pour placer un commentaire dans le document.

```
<!--ceci est un commentaire—>
```

À l'intérieur du commentaire l'utilisation des caractères -- est interdite :

```
<!--ceci est un -- commentaire-->
```

Caractères réservés

Certains caractères sont réservés par le langage XML comme, par exemple, le caractère & de l'exemple suivant :

```
<menu>fromage & dessert</menu>
```

Pour pouvoir utiliser ces caractères dans un document XML, vous devez les remplacer par la syntaxe suivante :

Caractère	utiliser à la place
&	&amp ;
<	&lt ;
>	&gt ;
'	&apos ;
"	&quot ;

La syntaxe correcte est donc :

```
<menu>fromage &amp; dessert</menu>
```

Des séquences de caractères plus longues peuvent être incorporées, en utilisant une section CDATA. La syntaxe est la suivante :

```
<![CDATA[{ Select * from desserts where prix < 10} and calories > 500]]>
```

Avec cette syntaxe, n'importe quel caractère peut être utilisé sans précaution particulière.

Éléments XML

Un élément XML est un conteneur qui accueille des données et d'autres éléments. Il se compose d'une balise de début et d'une balise de fin. La syntaxe d'un élément XML est la suivante :

```
<NomElement>contenu</NomElement>
```

Les éléments doivent respecter certaines règles concernant leur forme :

- Les noms d'éléments ne peuvent pas contenir d'espace.
- Ils ne peuvent pas débuter par un nombre ou un signe de ponctuation.
- Ils ne peuvent pas débuter par xml (quelle que soit la casse).
- Ils doivent débuter juste après le signe, sans espace.
- Les balises de début et de fin doivent avoir la même casse.
- Un document XML doit contenir au moins un élément : c'est l'élément racine.
- Tous les éléments qui suivent l'élément racine doivent être imbriqués dans celui-ci.
- Si un élément n'a pas de contenu, il peut être constitué uniquement par une balise de fin.
- Seul l'élément racine doit avoir une balise de début et une balise de fin, même s'il n'a pas de contenu.

Exemple :

```
<?xml version="1.0" encoding="utf-8" ?>
<restaurant>
<menu>
 <entrees>
   <nom>radis</nom>
   <nom>pate</nom>
   <nom>saucisson</nom>
 </entrees>
 <plats>
   <nom>choucroute</nom>
   <nom>cassoulet</nom>
   <nom>couscous</nom>
 </plats>
 <fromages>
   <nom>camembert</nom>
   <nom>brie</nom>
   <nom>roquefort</nom>
 </fromages>
 <desserts>
   <nom>glace</nom>
   <nom>tarte</nom>
   <nom>creme brule</nom>
 </desserts>
</menu>
</restaurant>
```

Attributs d'éléments

Les attributs d'éléments sont utilisés pour qualifier un élément. Ils sont placés dans la balise de début de l'élément. Comme les éléments, ils doivent suivre certaines règles :

- Un attribut est constitué d'un nom et d'une affectation de valeur.
- Un élément peut contenir un nombre quelconque d'attributs.
- Les noms d'attributs sont séparés par des espaces.
- Un nom d'attribut ne peut apparaître qu'une seule fois dans un élément.
- Un nom d'attribut peut apparaître dans plusieurs éléments.
- Un nom d'attribut ne peut pas contenir d'espace.

- L'affectation d'une valeur à un attribut se fait avec le signe égal suivi de la valeur entourée de doubles cotes.

Exemple :

```
<?xml version="1.0" encoding="utf-8" ?>
<restaurant>
<menu type="gastronomique">
  <entrees>
    <nom calories="50">radis</nom>
    <nom calories="300">pate</nom>
    <nom calories="350">saucisson</nom>
  </entrees>
  <plats>
    <nom calories="1000">choucroute</nom>
    <nom calories="2000">cassoulet</nom>
    <nom calories="1700">couscous</nom>
  </plats>
  <fromages>
    <nom calorie="240">camembert</nom>
    <nom calories="300">brie</nom>
    <nom calories="120">roquefort</nom>
  </fromages>
  <desserts>
    <nom calories="340" parfum="chocolat">glace</nom>
    <nom calories="250" fruits="pommes">tarte</nom>
    <nom calories="400">creme brule</nom>
  </desserts>
</menu>
</restaurant>
```

Espaces de noms

Un espace de nom est un ensemble de noms d'éléments identifiés par une référence unique. Ils permettent d'éviter les confusions lorsque des données XML sont fusionnées à partir de différentes sources.

Si nous prenons l'exemple suivant qui pourrait être un fichier de configuration d'une application :

```
<?xml version="1.0" encoding="utf-8" ?>
<application>
  <menu nom="fichier">
    <entrees>nouveau</entrees>
    <entrees>ouvrir</entrees>
    <entrees>fermer</entrees>
  </menu>
  <menu nom="edition">
    <entrees>copier</entrees>
    <entrees>couper</entrees>
    <entrees>coller</entrees>
  </menu>
</application>
```

Dans ce fichier, nous avons des éléments déjà définis dans un autre fichier. Il est clair que les éléments **menu** et **entrees** n'ont pas la même signification que dans le fichier utilisé précédemment. Pour éviter toute ambiguïté, il faut ajouter dans chacun des fichiers une définition d'espace de nom rendant unique chaque élément. La définition d'un espace de nom s'effectue par l'attribut `xmlns` suivi d'un préfixe et de l'identifiant de l'espace de nom.

La syntaxe est la suivante pour chacun de nos deux fichiers :

```
<restaurant xmlns:resto="http://www.eni-ecole.fr/restaurant">

<application xmlns:appli="http://www.eni-ecole.fr/configappli">
```

Il est très important que les identifiants d'espaces de nom soient uniques, si vous souhaitez échanger des informations avec d'autres personnes. C'est pourquoi, il est courant d'utiliser le nom de domaine de l'entreprise dans l'identifiant (celui-ci étant supposé unique). Avec cette modification, nous pouvons utiliser dans le même fichier des éléments **menu** et **entrees**, en ajoutant devant le préfixe de l'espace de nom dans lequel ils ont une signification.

```
<fusion xmlns:appli="http://www.eni-ecole.fr/configappli" xmlns:resto=
"http://www.eni-ecole.fr/restaurant">
  <appli:menu nom="fichier">
    <appli:entrees>enregistrer</appli:entrees>
  </appli:menu>
  <resto:menu nom="economique">
    <resto:entrees>avocat</resto:entrees>
  </resto:menu>
</fusion>
```

2. Document bien formé et document valide

Grâce au langage XML, nous avons la possibilité de créer facilement des documents structurés et compréhensibles. Il existe également deux notions permettant de vérifier la qualité d'un document XML : un document peut être bien formé et un document peut être valide.

a. Document bien formé

Un document est bien formé s'il obéit aux règles syntaxiques du langage XML. Ces règles sont beaucoup moins strictes que les règles de validité. Elles gèrent les attributions de noms, les créations et les imbrications d'éléments. Pour pouvoir être traité par un processeur XML, un document doit être bien formé. Si le processeur détecte une erreur, il arrête immédiatement le traitement du document.

b. Document valide

Un document valide est un document XML auquel est liée une DTD (définition du type de document) et qui respecte toutes les règles de construction définies dans cette dernière. Lorsqu'un processeur XML analyse le document, il recherche dans la DTD une définition pour tout élément, attribut, entité de ce document. Dès qu'il rencontre une erreur, il arrête le traitement.

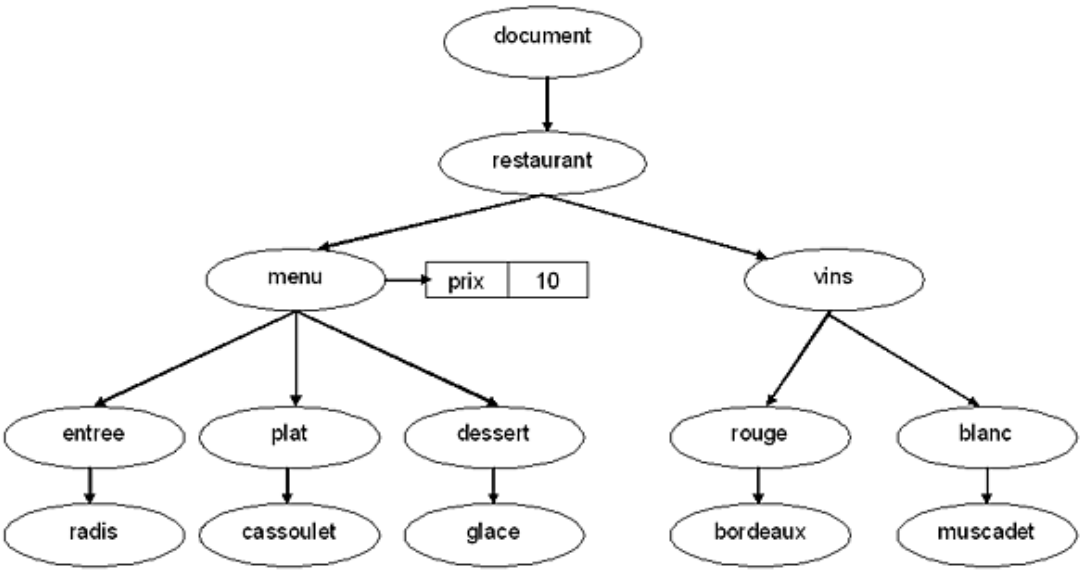
Manipulation d'un document XML

La manipulation d'un document XML dans une application VB.NET est facilitée par l'utilisation de DOM (*Document Object Model*). Le DOM vous permet de lire, de manipuler et de modifier un document XML par programme. Ce dernier régit la représentation en mémoire des données XML, bien que les données XML véritables soient stockées de façon linéaire lorsqu'elles se trouvent dans un fichier ou qu'elles proviennent d'un autre objet.

Par exemple, le document suivant :

```
<?xml version="1.0"?>
  <restaurant>
    <menu prix="10">
        <entree>radis</entree>
        <plat>cassoulet</plat>
        <dessert>glace</dessert>
    </menu>
    <vins>
        <rouge>bordeaux</rouge>
        <blanc>muscadet</blanc>
    </vins>
  </restaurant>
```

est représenté sous cette forme en mémoire dans une application :



Dans la structure d'un document XML, chaque cercle de cette illustration représente un nœud, appelé objet `XmlNode` qui est l'objet de base de l'arborescence DOM. La classe`XmlDocument` prend en charge des méthodes destinées à exécuter des opérations sur le document dans son ensemble, par exemple pour le charger en mémoire ou l'enregistrer sous la forme d'un fichier. Les objets `XmlNode` comportent un ensemble de méthodes et de propriétés, ainsi que des caractéristiques de base bien définies. Voici certaines de ces caractéristiques :

- Un nœud ne possède qu'un seul nœud parent, qui est le nœud situé juste au-dessus de lui.
- Le seul nœud qui est dépourvu de parent est la racine du document, puisqu'il s'agit du nœud de premier niveau qui contient le document lui-même et les fragments de document.
- La plupart des nœuds peuvent comporter plusieurs nœuds enfants, qui sont les nœuds situés directement sous eux.
- Les nœuds situés au même niveau, représentés dans le diagramme par les nœuds menu et vins, sont des nœuds frères.

L'une des caractéristiques du DOM est la manière dont il gère les attributs. Les attributs ne sont pas des nœuds qui font partie des relations parent-enfant et frère. Ils sont considérés comme une propriété du nœud et sont constitués d'une paire, composée d'un nom et d'une valeur.

Dans notre exemple, prix="10" associé à l'élément menu, le mot prix correspond au nom et la valeur de l'attribut prix est 10. Pour extraire l'attribut prix="10" du nœud menu, vous appelez la méthode `GetAttribute` lorsque le curseur se trouve sur le nœud menu.

Pour les exemples qui suivent, nous utiliserons le document XML suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<restaurant>
  <menu type="gastronomique">
    <entrees>
      <nom calories="50">radis</nom>
      <nom calories="300">pate</nom>
      <nom calories="350">saucisson</nom>
    </entrees>
    <plats>
      <nom calories="1000">choucroute</nom>
      <nom calories="2000">cassoulet</nom>
      <nom calories="1700">couscous</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
      <nom calories="300">brie</nom>
      <nom calories="120">roquefort</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="chocolat">glace</nom>
      <nom calories="250" fruits="pommes">tarte</nom>
      <nom calories="400">creme brule</nom>
    </desserts>
  </menu>
  <menu type="economique">
    <entrees>
      <nom calories="50">pain</nom>
    </entrees>
    <plats>
      <nom calories="1700">jambon</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="a l'eau">glace</nom>
    </desserts>
  </menu>
</restaurant>
```

1. Utilisation de DOM

La première étape, lors de l'utilisation de DOM, consiste à charger le document XML dans un arbre de nœuds DOM. Vous devez pour cela déclarer un objet `XmlDocument` puis utiliser la méthode `Load` pour remplir cet objet à partir d'un fichier XML.

```
dim as XmlDocument
doc = New Xml.XmlDocument()
doc.Load("resto.xml")
```

Il est également possible de charger des données XML à partir d'une chaîne de caractères. Vous devez, dans ce cas, utiliser la méthode `LoadXML` en lui fournissant la chaîne de caractères contenant les données XML.

Une fois les données XML chargées dans l'arbre, vous pouvez localiser des nœuds particuliers afin de les soumettre à des opérations de traitement, de manipulation ou de modification. La méthode `GetElementsByTagName` permet d'obtenir un objet `XmlNodeList` contenant les nœuds concernés. Vous pouvez alors obtenir les attributs du nœud en utilisant la propriété `Attributes` ou vérifier s'ils possèdent des nœuds enfants avec la propriété `HasChildNodes`. Si c'est le cas, vous avez accès à ces nœuds à l'aide de la propriété `ChildNodes` sous forme d'un objet `XmlNodeList`.

L'exemple suivant recherche les nœuds menu dans l'arboresence et affiche l'attribut `type`.

```
menus = doc.GetElementsByTagName("menu")
```

```
        For Each unMenu In menus
            Console.WriteLine(unMenu.Attributes("type").Value)
        Next
```

Les caractéristiques des nœuds peuvent également être modifiées en y ajoutant un attribut. Les nœuds menu peuvent par exemple recevoir un attribut **prix**.

```
menus = doc.GetElementsByTagName("menu")
        Dim att As XmlAttribute
        For Each unMenu In menus
            If unMenu.Attributes("type").Value = "gastronomique" Then
                att = doc.CreateAttribute(prix")
                att.Value = "50 "
                unMenu.Attributes.Append(att)
            End If
            If unMenu.Attributes("type").Value = "economique" Then
                att = doc.CreateAttribute(prix")
                att.Value = "15 "
                unMenu.Attributes.Append(att)
            End If
        Next
```

Il est également possible d'ajouter des nœuds enfants à des nœuds existant dans l'arboresence, en créant des instances de la classe `XmlNode` et en les reliant à leur nœud parent. L'exemple suivant ajoute un **digestif** au menu **gastronomique**.

```
menus = doc.GetElementsByTagName("menu")
        Dim att As XmlAttribute
        For Each unMenu In menus
            If unMenu.Attributes("type").Value = "gastronomique" Then
                Dim n1, n2, n3 As XmlNode
                n1 = doc.CreateNode(XmlNodeType.Element, "digestif", "")
                n2 = doc.CreateNode(XmlNodeType.Element, "nom", "")
                n3 = doc.CreateNode(XmlNodeType.Text, “”, “”)
                n3.Value = "Cognac"
                n2.AppendChild(n3)
                n1.AppendChild(n2)
                unMenu.AppendChild(n1)
            End If
        Next
```

Après l'exécution des deux exemples précédents, le document XML doit avoir la forme suivante :

```
<?xml version="1.0" encoding="Windows-1252"?>
<restaurant>
  <menu type="gastronomique" prix="50 <$&euro[-]>">
    <entrees>
      <nom calories="50">radis</nom>
      <nom calories="300">pate</nom>
      <nom calories="350">saucisson</nom>
    </entrees>
    <plats>
      <nom calories="1000">choucroute</nom>
      <nom calories="2000">cassoulet</nom>
      <nom calories="1700">couscous</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
      <nom calories="300">brie</nom>
      <nom calories="120">roquefort</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="chocolat">glace</nom>
      <nom calories="250" fruits="pommes">tarte</nom>
      <nom calories="400">creme brule</nom>
    </desserts>
    <digestif>
      <nom>Cognac</nom>
    </digestif>
  </menu>
```

```
  <menu type="economique" prix="15 <$&euro[-]>">
    <entrees>
      <nom calories="50">pain</nom>
    </entrees>
    <plats>
      <nom calories="1700">jambon</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="a l'eau">glace</nom>
    </desserts>
  </menu>
</restaurant>
```

En fait, seule la représentation en mémoire du document XML est modifiée. Si vous souhaitez conserver les modifications, il faut enregistrer le document dans un fichier pour assurer la persistance des informations. Pour cela, vous devez utiliser la méthode `save` de la classe `XmlDocument`, en lui fournissant le nom du fichier dans lequel vous souhaitez effectuer la sauvegarde.

```
doc.Save("resto2.xml")
```

2. Utilisation de XPath

L'objectif principal de XPath est de définir la manière d'adresser des parties d'un document XML. Le nom XPath vient de l'utilisation d'une écriture de type "path", comme dans les shells DOS et UNIX. Le but est de se déplacer à l'intérieur de la structure hiérarchique d'un document XML comme dans une arborescence de répertoires. Pour avoir une idée de l'intérêt de XPath, nous pourrions dire qu'il est l'équivalent du langage SQL pour un document XML. La comparaison doit s'arrêter là car la syntaxe n'a vraiment rien à voir !

a. Recherche dans un document XML

La première étape, pour rechercher un élément dans document XML, consiste à créer une instance de la classe`XPathNavigator`. Cette instance de classe doit connaître le document sur lequel elle va devoir faire des recherches, c'est pourquoi c'est le document lui-même qui par l'intermédiaire de la méthode `CreateNavigator` va fournir cette instance de classe.

```
Dim navigateur As XPathNavigator = document.CreateNavigator()
```

À partir de cette instance, nous allons pouvoir lancer des recherches dans le document à l'aide de la méthode `Select`. Cette méthode utilise comme paramètre une chaîne de caractères contenant le chemin `XPath` de recherche. Nous obtenons, après l'exécution, un objet `XPathNodeIterator` permettant de parcourir la liste des nœuds trouvés.

L'exemple suivant recherche dans le document **resto.xml**, les entrées disponibles dans les différents menus :

```
Dim document As XmlDocument = New XmlDocument ()
 document.Load("resto.xml")

 Dim navigateur As XPathNavigator = document.CreateNavigator()
 Dim noeuds As XPathNodeIterator = navigateur.Select("/restaurant/menu/entrees")
 While noeuds.MoveNext()
            Console.WriteLine(noeuds.Current.OuterXml)
            Console.WriteLine()
 End While
```

Nous obtenons le résultat suivant :

```
<entrees>
  <nom calories="50">radis</nom>
  <nom calories="300">pate</nom>
  <nom calories="350">saucisson</nom>
</entrees>

<entrees>
  <nom calories="50">pain</nom>
```

```
</entrees>
```

Il est également possible d'ajouter à la requête XPath des critères de sélection sur la valeur de certains attributs.

L'exemple suivant recherche les desserts du menu **gastronomique** pour lesquels les calories sont inférieures à **350**.

```
Dim document As XmlDocument = New XmlDocument ()
 document.Load("resto.xml")
 Dim navigateur As XPathNavigator = document.CreateNavigator()
 Dim noeuds As XPathNodeIterator = navigateur.Select("/restaurant/menu[@type=
'gastronomique']/desserts/nom[@calories^]")
        While noeuds.MoveNext()
            Console.WriteLine(noeuds.Current.Value)
            Console.WriteLine()
        End While
```

b. Modification des données d'un document XML

Après avoir trouvé un élément dans l'arborescence d'un document, il est bien sûr possible d'en modifier sa valeur.

L'exemple suivant diminue de **50%** les calories de chaque dessert du menu **gastronomique**.

```
        Dim document As XmlDocument = New XmlDocument()
        document.Load("resto.xml")
        Dim navigateur As XPathNavigator = document.CreateNavigator()
        Dim noeuds As XPathNodeIterator = navigateur.Select("/restaurant/menu
[ @type='gastronomique']/desserts/nom")

        While noeuds.MoveNext()
            noeuds.Current.MoveToAttribute("calories", "")
            noeuds.Current.SetValue(noeuds.Current.Value * 0.5)
        End While
document.Save("resto.xml")
```

Voici le contenu du fichier après exécution du code précédent.

```
<?xml version="1.0" encoding="utf-8"?>
<restaurant>
  <menu type="gastronomique">
    <entrees>
      <nom calories="50">radis</nom>
      <nom calories="300">pate</nom>
      <nom calories="350">saucisson</nom>
    </entrees>
    <plats>
      <nom calories="1000">choucroute</nom>
      <nom calories="2000">cassoulet</nom>
      <nom calories="1700">couscous</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
      <nom calories="300">brie</nom>
      <nom calories="120">roquefort</nom>
    </fromages>
    <desserts>
      <nom calories="170" parfum="chocolat">glace</nom>
      <nom calories="125" fruits="pommes">tarte</nom>
      <nom calories="200">creme brule</nom>
    </desserts>
  </menu>
  <menu type="economique">
    <entrees>
      <nom calories="50">pain</nom>
    </entrees>
    <plats>
      <nom calories="1700">jambon</nom>
    </plats>
    <fromages>
```

```
      <nom calorie="240">camembert</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="a l'eau">glace</nom>
    </desserts>
  </menu>
</restaurant>
```

c. Ajout de nœud à un document XML

Après la recherche d'un nœud dans un document, il est possible d'y ajouter des nœuds enfants et des nœuds frères. Les méthodes `InsertAfter` et `InsertBefore` ajoutent un nœud frère après ou avant le nœud actuel. La méthode `AppendChild` ajoute un nœud enfant au nœud actuel.

L'exemple suivant ajoute un nouveau dessert au menu **gastronomique**.

```
Dim document As XmlDocument = New XmlDocument()
  document.Load("resto.xml")
  Dim navigateur As XPathNavigator = document.CreateNavigator()
 Dim noeuds As XPathNodeIterator = navigateur.Select("/restaurant/menu[@type=
' gastronomique']/desserts") noeuds.MoveNext()
 noeuds.Current.AppendChild("<nom calories='800'>crepes</nom>")
 document.Save("resto.xml")
```

Après l'exécution de ce code, le document devient :

```
<?xml version="1.0" encoding="utf-8"?>
<restaurant>
  <menu type="gastronomique">
    <entrees>
      <nom calories="50">radis</nom>
      <nom calories="300">pate</nom>
      <nom calories="350">saucisson</nom>
    </entrees>
    <plats>
      <nom calories="1000">choucroute</nom>
      <nom calories="2000">cassoulet</nom>
      <nom calories="1700">couscous</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
      <nom calories="300">brie</nom>
      <nom calories="120">roquefort</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="chocolat">glace</nom>
      <nom calories="250" fruits="pommes">tarte</nom>
      <nom calories="400">creme brule</nom>
      <nom calories="800">crepes</nom>
    </desserts>
  </menu>
  <menu type="economique">
    <entrees>
      <nom calories="50">pain</nom>
    </entrees>
    <plats>
      <nom calories="1700">jambon</nom>
    </plats>
    <fromages>
      <nom calorie="240">camembert</nom>
    </fromages>
    <desserts>
      <nom calories="340" parfum="a l'eau">glace</nom>
    </desserts>
  </menu>
</restaurant>
```

Introduction

Maintenant que votre application est terminée, testée, déboguée et donc qu'elle fonctionne sans problèmes, il est temps de penser au moyen de la mettre à la disposition des utilisateurs. Deux solutions sont disponibles :

- La technologie de déploiement Windows Installer. L'application est empaquetée dans un ou plusieurs fichiers qui sont ensuite distribués aux utilisateurs ; ceux-ci exécutent le fichier Setup.exe pour installer l'application.

- Le déploiement ClickOnce. Avec cette solution, la publication des fichiers de l'application se fait à un emplacement centralisé et l'utilisateur installe ou exécute l'application à partir de cet emplacement.

Nous allons donc détailler chacune de ces deux techniques de déploiement.

Déploiement avec Windows Installer

Microsoft Windows Installer est un service d'installation et de configuration d'application disponible sur tous les systèmes d'exploitation Microsoft. Le principe de base du fonctionnement de Windows Installer repose sur le regroupement dans un seul élément de toutes les données et instructions nécessaires pour le déploiement d'une application. C'est une évolution importante par rapport aux procédures d'installations classiques qui consistaient essentiellement à fournir l'ensemble des fichiers nécessaires pour le bon fonctionnement de l'application et un script chargé de la recopie de ces fichiers sur le disque dur de la machine.

Avec Windows Installer, le système conserve une trace de toutes opérations effectuées pendant l'installation : répertoires créés, fichiers copiés, entrées de la base de registre modifiées, etc. Ces informations sont par la suite utilisées lors de la désinstallation de l'application. Windows Installer effectue alors les opérations inverses pendant la désinstallation de l'application. Un contrôle est cependant réalisé pour s'assurer qu'aucune autre application ne nécessite un fichier, une clé de registre ou un composant qui s'apprête à être supprimé. Cette vérification permet de s'assurer que la suppression d'une application n'entraîne pas de problèmes de fonctionnement sur une autre application.

Windows Installer gère également la réparation d'une application en réinstallant automatiquement les fichiers manquants qui ont pu être supprimés, par mégarde, par l'utilisateur.

La procédure d'installation est effectuée à l'intérieur d'une transaction, garantissant que l'application sera installée complètement ou que, en cas d'échec au cours de l'installation, le système retrouvera son état initial.

Les procédures d'installations sont en fait de véritables applications, elles sont d'ailleurs gérées par Visual Studio comme des projets à part entière.

1. Création d'un projet d'installation

La méthode de création d'un projet d'installation est identique à celle utilisée pour n'importe quel autre type de projet de Visual Studio. Dans le menu **Fichier**, sélectionnez **Ajouter**, puis **Nouveau projet**. Dans la boîte de dialogue d'**Ajout de projet**, sélectionnez ensuite **Autres types de projets** et **Projets d'installation et de déploiement**. Enfin choisissez parmi les modèles celui qui correspond à vos besoins (**Projet d'installation** dans notre cas). Le projet est alors ajouté à la solution actuelle et l'éditeur du système de fichiers est ouvert automatiquement. Vous pouvez maintenant configurer les propriétés du programme d'installation qui vont déterminer son comportement. Voici ci-dessous les principales propriétés d'un programme d'installation :

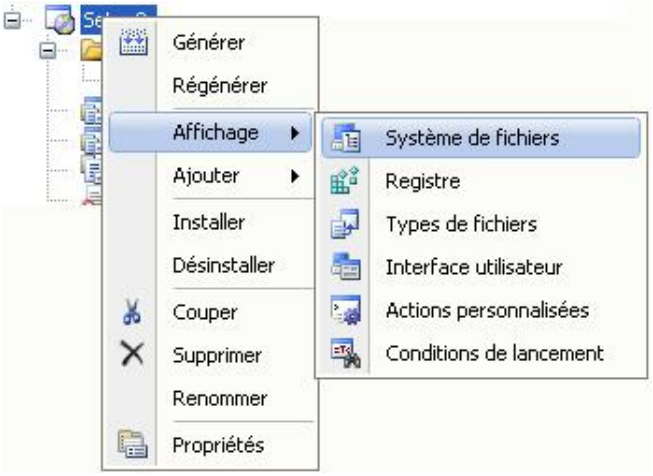
- **AddRemoveProgramsIcon** : indique l'icône utilisée pour représenter l'application dans la boîte de dialogue **Ajout/Suppression de programmes**.
- **Author** : spécifie le nom de l'auteur de l'application. Par défaut cette propriété correspond au nom du détenteur de la licence de Visual Studio utilisé pour développer l'application.
- **DetectNewerInstalledVersion** : indique si le programme d'installation vérifie l'existence d'une version plus récente de l'application sur le poste client. Si c'est le cas la procédure d'installation est annulée.
- **ManufacturerUrl** : adresse du site Web du fabricant de l'application. Elle est affichée dans la boîte de dialogue **Informations de support technique** accessible à partir de la boîte de dialogue **Ajout/Suppression de programmes**.
- **ProductCode** : Windows Installer utilise cette propriété pour identifier une application lors de chaque installation ou mise à jour ; deux applications différentes ne peuvent pas comporter le même code ProductCode.
- **ProductName** : contient le nom décrivant l'application qui est installée sur un ordinateur cible. Par défaut, ce nom est identique à celui du projet de déploiement. Cette propriété `ProductName` est affichée dans la description de l'application sur la boîte de dialogue **Ajout/Suppression de programmes**. Elle est également utilisée dans la création du chemin d'installation par défaut utilisé lors de l'installation.
- **RemovePreviousVersions** : spécifie si le programme d'installation doit supprimer, lors de l'installation, toute version antérieure d'une application. Si cette propriété a la valeur True et qu'un numéro de version antérieure est détecté lors de l'installation, la fonction de désinstallation de cette version est appelée.
- **TargetPlatform** : spécifie la plate-forme matérielle pour laquelle l'application a été conçue. Au moment de l'installation, cette propriété est vérifiée par rapport à l'ordinateur cible pour déterminer si l'installation peut continuer.

- **Version** : spécifie le numéro de version du programme d'installation. Cette propriété doit être modifiée pour chaque version finale de votre programme d'installation. Si vous modifiez cette propriété, vous devez également mettre à jour la propriété `ProductCode`.

Un petit détail doit vous sembler bizarre car parmi toutes ces propriétés aucune d'entre elles ne permet d'indiquer quelle application doit être installée. Cette information doit en fait être fournie lors de la configuration du programme d'installation. C'est ce que nous allons voir dans l'étape suivante.

2. Configuration du programme d'installation

La configuration du fonctionnement du programme d'installation se fait par l'intermédiaire d'un éditeur spécifique à chacune des fonctionnalités. Ces éditeurs sont disponibles par le menu contextuel affiché en effectuant un clic droit sur le nom du projet de déploiement dans l'explorateur de solutions.



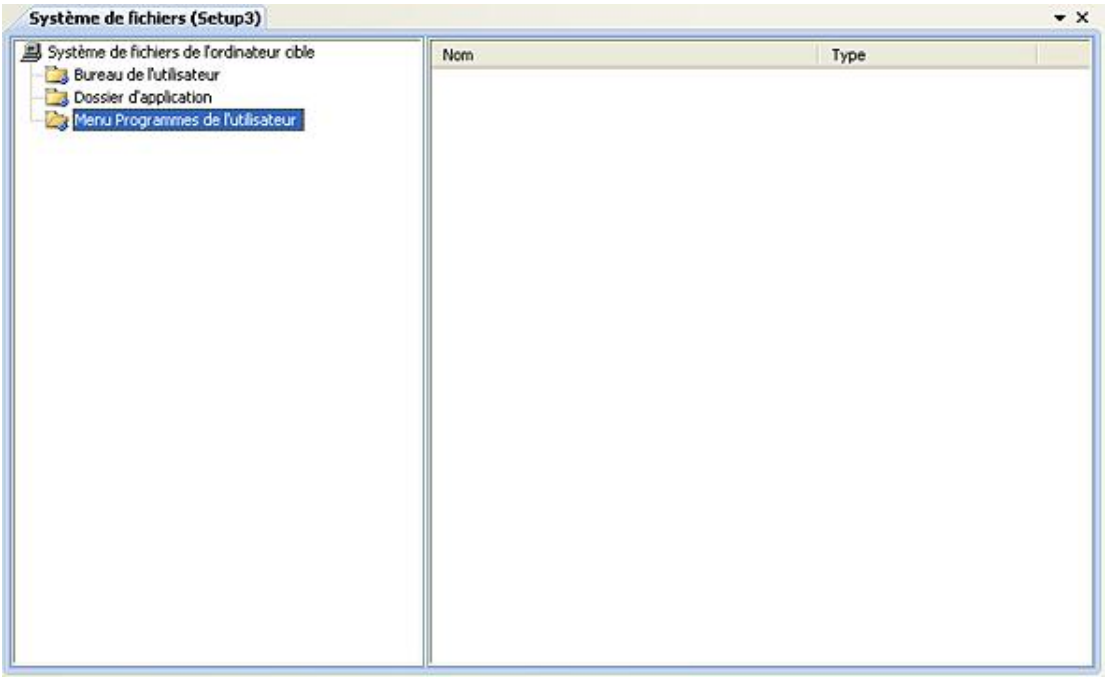
Ces éditeurs vont permettre de configurer le fonctionnement du programme d'installation.

- L'éditeur du système de fichiers va permettre d'ajouter à un projet de déploiement les fichiers constituant l'application à déployer.
- L'éditeur du registre configure les clés et les valeurs de Registre à ajouter au Registre de l'ordinateur cible.
- L'éditeur des types de fichiers spécifie les associations entre les types de fichiers et les actions autorisées pour chaque type de fichier.
- L'éditeur de l'interface utilisateur permet de spécifier et de configurer les boîtes de dialogue affichées pendant l'installation de l'application.
- L'éditeur des actions personnalisées permet de spécifier les actions supplémentaires à exécuter pendant l'installation.
- L'éditeur des conditions de lancement spécifie les conditions requises pour l'installation.

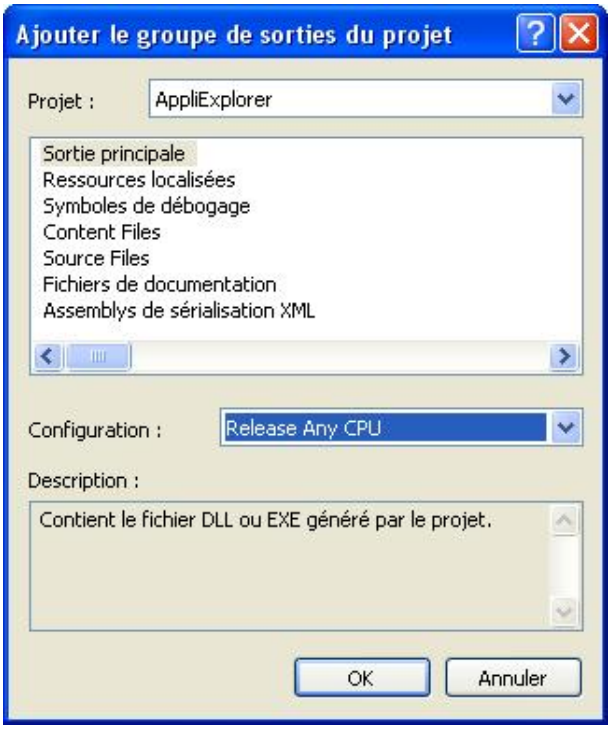
Nous allons étudier en détail chacun de ces éditeurs.

a. L'éditeur du système de fichiers

C'est certainement l'éditeur le plus important de la configuration d'un projet de déploiement. Par son intermédiaire, nous allons indiquer les modifications à apporter au système de fichiers de l'ordinateur sur lequel l'application va être déployée.

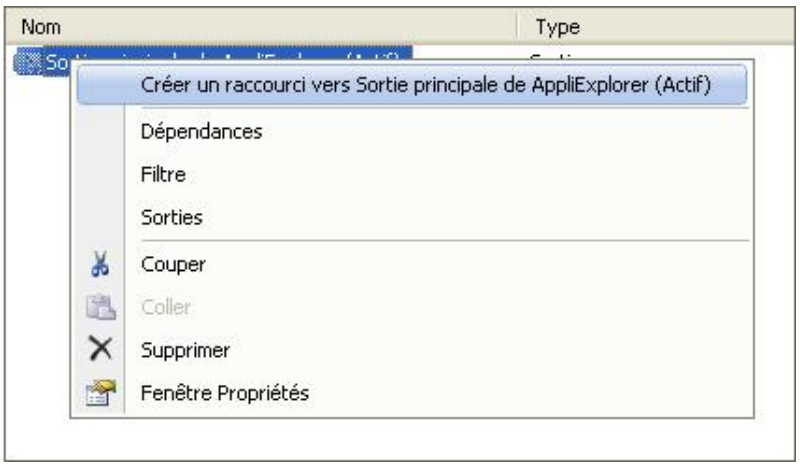


Il est composé de deux parties : un volet de navigation à gauche et un volet d'informations à droite. Le volet de navigation contient la liste hiérarchique des dossiers correspondant au système de fichiers de l'ordinateur d'installation. Les noms des dossiers correspondent aux dossiers Windows standard. Par exemple, la rubrique dossier d'application désigne le sous-dossier du dossier **Program Files** où l'application sera installée. Le menu contextuel de chacun des dossiers permet d'y ajouter différents éléments. Dans le dossier d'application, nous pouvons par exemple ajouter un sous-dossier, un fichier, un assembly ou, option la plus fréquemment utilisée, une sortie de projet (les fichiers créés par la compilation d'un projet). C'est donc grâce à cet éditeur que nous allons enfin pouvoir indiquer quel projet nous souhaitons déployer. La boîte de dialogue suivante nous permet de choisir le projet à déployer et lesquels de ces éléments nous souhaitons déployer sur les postes des clients.



Pour que l'application puisse s'exécuter sur l'ordinateur client, il faut au minimum sélectionner l'option **Sortie principale**. Par contre, l'option **Source Files** est plus rarement utilisée. Vous devez également indiquer si vous souhaitez déployer la version Debug ou la version Release de l'application. L'ajout de plusieurs éléments se fait en maintenant la touche [Ctrl] enfoncée pendant la sélection.

Les dossiers User's Desktop et User's Program Menu reçoivent généralement un simple raccourci vers l'application. Ce raccourci doit au préalable être créé par le menu contextuel de la sortie principale de l'application.

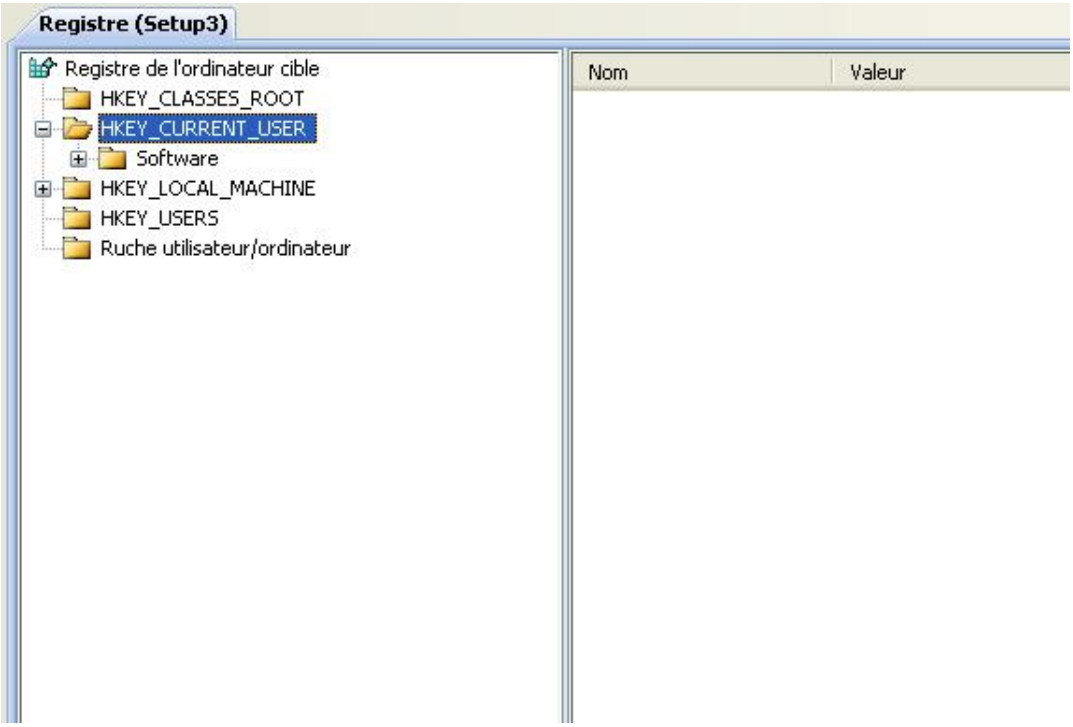


Il doit ensuite être déplacé vers le dossier correspondant (Bureau de l'utilisateur ou Menu Programmes de l'utilisateur).

Pour que chacun de ces dossiers soient créés même s'il ne contient aucun élément, il faut modifier sa propriété `AlwaysCreate` sur True.

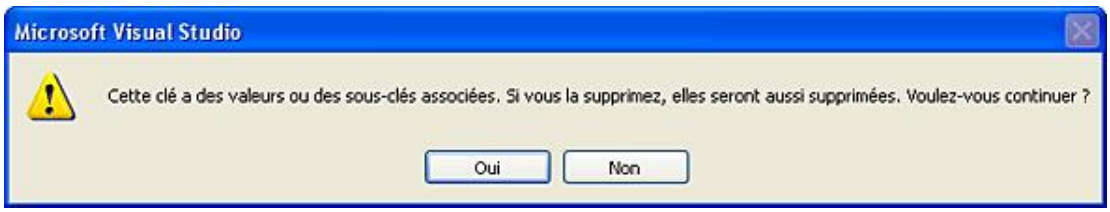
b. L'éditeur du registre

L'éditeur du Registre permet d'ajouter des clés de Registre à un projet de déploiement. Si une clé n'existe pas dans le registre de la machine lors du déploiement, elle est ajoutée pendant l'installation. Il est possible d'ajouter des clés sous n'importe quelle clé de niveau supérieur dans l'éditeur du registre.



Pour ajouter une clé dans le registre, vous devez au préalable sélectionner un nœud de niveau supérieur, ou une sous-clé puis à l'aide du menu contextuel utiliser l'option **New - Key**. La clé doit ensuite être renommée et ses propriétés modifiées en fonction des besoins. La principale modification consiste à configurer la propriété `DeleteAtUninstall` pour que cette clé de registre soit supprimée lors de la désinstallation de l'application.

La suppression d'une clé est réalisée tout aussi simplement avec l'option **Supprimer** du menu contextuel. Il faut cependant être prudent car la suppression d'une clé entraîne la suppression de toutes les sous-clés et valeurs contenues dans celle-ci. Un message d'avertissement est affiché pour vous prévenir de cette situation dangereuse et vous demande de confirmer votre choix.



L'éditeur du registre est également utilisé pour spécifier les valeurs des nouvelles clés ou modifier les valeurs existantes. Vous pouvez ajouter des valeurs de type chaîne, binaire et DWORD. Pendant l'installation, les valeurs sont écrites dans le Registre ; les valeurs existantes sont remplacées par les valeurs spécifiées dans le programme d'installation.

Il est possible d'ajouter des clés et des valeurs de registre à un projet de déploiement en important un fichier de registre (.reg) dans l'éditeur du registre. Cela vous permettra de copier une section complète d'un registre existant en une seule fois pour gagner du temps. Les fichiers de registre peuvent être créés à l'aide d'outils tels que l'éditeur du registre de Windows (regedit.exe). Cette solution est très pratique pour transférer sur le poste des utilisateurs une portion de registre récupérée sur le poste utilisé pour le développement de l'application. Le menu contextuel disponible sur l'élément Registre de l'ordinateur cible propose l'option Importer permettant de réaliser cette opération. Vous devez simplement choisir le fichier (.reg) contenant les informations à importer.

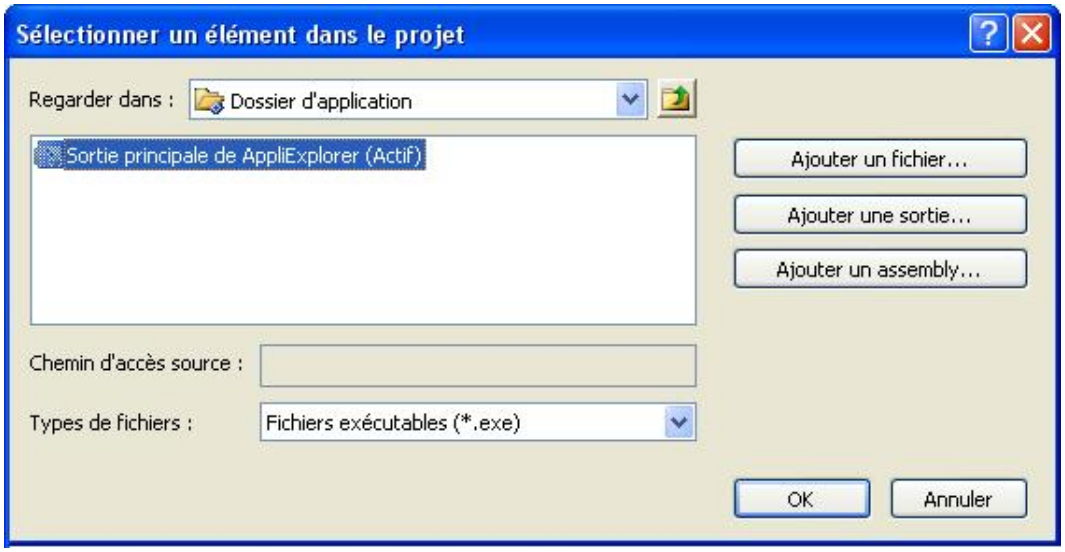
c. L'éditeur des types de fichiers

L'éditeur des types de fichiers est utilisé pour indiquer les types de documents et les extensions de fichier associés à votre application lors de son installation sur un ordinateur. Une fois l'association installée, l'extension et la description du type de fichier figurent dans la liste des types de fichiers du système.

L'association d'un type de fichier avec une application nécessite trois étapes :

- L'ajout d'un type de document.
- L'association d'une extension de fichier.
- L'association d'un fichier exécutable.

Pour ajouter un type de fichier vous devez utiliser le menu contextuel de l'éditeur des types de fichiers et choisir l'option **Ajouter un type de fichier**. Il faut ensuite modifier le nom de l'élément qui vient d'être ajouté. L'étape suivante consiste à indiquer l'extension ou les extensions associées à ce type de fichier. Elles doivent être saisies dans la fenêtre de propriétés, dans la rubrique **Extensions**, sans être précédées d'un point. Si plusieurs extensions sont disponibles, elles doivent être séparées par un point virgule lors de la saisie dans la fenêtre de propriétés. La dernière étape est maintenant d'associer une application (un fichier exécutable) à ce type de fichier. La propriété `Command` est utilisée à cet effet. Un éditeur particulier vous permet de rechercher parmi les dossiers du projet l'exécutable qui sera utilisé pour manipuler ce type de document.



C'est cet exécutable qui sera lancé lorsque l'utilisateur effectuera un double clic sur un fichier de ce type dans l'Explorateur Windows. Il est possible d'ajouter d'autres actions qui seront disponibles par l'intermédiaire du menu contextuel de l'explorateur Windows. Ces éléments peuvent être ajoutés en utilisant le menu contextuel disponible sur un type de fichier et en choisissant l'option **Ajouter une action**. L'action doit ensuite être configurée en indiquant

les paramètres suivants :

- nom de l'action : utilisé pour représenter l'action dans le menu contextuel de l'explorateur Windows. Pour inclure une touche d'accès rapide dans le nom de l'action, faites précéder du signe & la lettre utilisée pour accéder à la commande.

- Arguments : représente les paramètres passés sur la ligne de commande permettant la réalisation de l'action. Par exemple, la chaîne suivante peut être associée à l'action d'impression d'un document.: /p "%1". Dans cette chaîne de caractères, la partie "%1" représente le nom du fichier sur lequel le menu contextuel a été activé. Les caractères " sont obligatoires pour éviter les problèmes si le nom de fichier comporte un espace. Il faut bien sur que votre application ait été conçue pour traiter ces paramètres lors de son démarrage.

- Verb : indique le verbe utilisé pour demander l'exécution de l'action.

d. L'éditeur de l'interface utilisateur

Avec l'Éditeur de l'interface utilisateur, vous pouvez spécifier et définir les propriétés des boîtes de dialogue prédéfinies qui seront affichées pendant l'installation sur l'ordinateur.

Cet éditeur contient deux sections : `Install` et `Administrative Install`. La section `Install` contient les boîtes de dialogue qui seront affichées quand l'utilisateur final exécutera le programme d'installation ; la section `Administrative Install` contient des boîtes de dialogue qui seront affichées quand un administrateur système téléchargera le programme d'installation vers un emplacement réseau.

Un ensemble de boîtes de dialogue prédéfinies est affiché dans l'éditeur ; vous pouvez les réorganiser ou les supprimer.

Les boîtes de dialogue prédéfinies se répartissent en trois catégories :

- Les boîtes de dialogue **Début** s'affichent avant que l'installation ne commence. Elles servent généralement à récupérer des informations sur l'utilisateur ou permettre de changer de répertoire d'installation.

- Une boîte de dialogue **Progression** donne des informations sur l'avancement d'une installation.

- Les boîtes de dialogue **Fin** s'affichent lorsque l'installation a réussi. Elles servent généralement à signaler à l'utilisateur que l'installation est terminée ou à lui permettre de lancer l'application.

Vous pouvez déplacer des boîtes de dialogue entre les nœuds des catégories par un glisser-déplacer ou à l'aide des commandes **Couper** et **Coller** du menu **Edition**.

Pour que l'installation se fasse sans boîtes de dialogue vous devez supprimer toutes les boîtes de dialogue dans l'Éditeur de l'interface utilisateur.

Chaque boîte de dialogue peut être personnalisée en fonction des besoins. Pour chacune d'entre elles les propriétés disponibles modifient l'aspect de la boîte de dialogue. Sur toutes ces boîtes de dialogue, la propriété `BannerBitmap` représente le logo affiché sur la boîte de dialogue. Des propriétés spécifiques sont accessibles sur chacune des boîtes de dialogue.

Boîte de dialogue Bienvenue

- La propriété `WelcomeText` contient le message de présentation de l'application.

- La propriété `CopyrightWarning` indique les informations relatives au copyright de l'application.

Boîte de dialogue Dossier d'installation

- La propriété `InstallAllUsersVisible` détermine si les boutons permettant de choisir le type d'installation (pour un seul utilisateur ou pour tous les utilisateurs) sont visibles.

Boîte de dialogue Progression

- L'affichage d'une barre de progression au cours de l'installation est déterminé par la propriété `ShowProgressBar`.

Boîte de dialogue Terminé

Les informations concernant les mises à jour disponibles sont affichées par la propriété `UpdateText`.

e. L'éditeur des actions personnalisées

Cet éditeur permet de spécifier des actions supplémentaires à exécuter sur l'ordinateur pendant l'installation. Par exemple, il peut être utile d'exécuter un programme créant une base de données sur un serveur.

Avant de pouvoir être ajoutées à un projet de déploiement, les actions personnalisées doivent être compilées sous forme de fichier .dll ou .exe, ou ajoutées à un projet en tant que script ou assembly. Elles ne peuvent être exécutées qu'à la fin de l'installation.

L'éditeur contient quatre dossiers qui correspondent chacun à une phase de l'installation :

Installer

Les actions personnalisées placées sous ce nœud seront exécutées à la fin de la phase d'installation, une fois tous les fichiers installés.

Valider

Les actions personnalisées placées sous ce nœud seront exécutées à la fin de la phase de validation de l'installation, qui a lieu une fois la phase d'installation terminée sans erreur.

Restaurer

Les actions personnalisées placées sous ce nœud seront exécutées à la fin de la phase de restauration de l'installation, qui est déclenchée lorsqu'une erreur d'installation se produit.

Désinstaller

Les actions personnalisées placées sous ce nœud seront exécutées à la fin de la phase de désinstallation de l'installation, qui a lieu lorsqu'une application est désinstallée.

Pour chacune de ces rubriques vous pouvez ajouter une action personnalisée par l'option **Add Custom Action** du menu contextuel. Vous devez ensuite sélectionner parmi les éléments disponibles dans le projet celui chargé de réaliser votre action personnalisée (.exe, .dll , .vbs…).

Les actions personnalisées seront exécutées dans leur ordre d'affichage dans l'éditeur. Vous pouvez modifier cet ordre par un glisser-déplacer ou à l'aide des commandes **Couper** et **Coller** du menu **Edition**.

f. L'éditeur des conditions de lancement

Cet éditeur doit être utilisé pour spécifier les conditions requises pour l'installation de votre application. Les conditions peuvent concerner la recherche d'un fichier, d'une entrée du registre ou d'un composant. Le résultat de la recherche est retourné sous forme d'une propriété qui est ensuite évaluée durant l'installation. L'installation échoue si une condition n'est pas remplie.

L'éditeur des conditions de lancement est constitué d'une arborescence composée de deux sections : la section **Recherche de l'ordinateur cible pour configurer les recherches** et la section **Conditions de lancement pour établir les conditions basées sur le résultat des recherches**.

Dans un premier temps vous devez ajouter, par le menu contextuel de la rubrique **Recherche de l'ordinateur cible**, les recherches à effectuer avant l'installation.

Recherche sur un fichier

Pour cette catégorie de recherche, les propriétés suivantes sont disponibles :

- `Depth` : spécifie le nombre de niveaux de sous-dossiers dans lesquels rechercher un fichier. La valeur par défaut est 0 indiquant que seul le dossier indiqué sera parcouru.
- `FileName` : spécifie le nom du fichier à rechercher. Les caractères génériques (* ?) ne sont pas autorisés.
- `Folder` : spécifie le dossier dans lequel la recherche va débuter. Les noms des dossiers spéciaux reconnus par Windows Installer doivent être saisie entre les caractères [et]. La valeur par défaut est [SystemFolder].

- `MaxDate` : le fichier recherché ne doit pas être plus récent que cette date.
- `MaxSize` : spécifie la taille maximale (en octets) du fichier recherché. Elle ne doit pas être supérieure à cette valeur.
- `MaxVersion` : spécifie le numéro de version maximal d'un fichier.
- `MinDate` : le fichier recherché doit être plus récent que la date indiquée.
- `MinSize` : spécifie la taille minimale (en octets) d'un fichier lors d'une recherche de fichiers. Elle doit être supérieure à cette valeur.
- `MinVersion` : spécifie le numéro de version minimal d'un fichier. Ce numéro doit être saisi avec le format n.n.n.n, où n est un entier.
- `Name` : spécifie le nom utilisé dans l'Éditeur des conditions de lancement pour identifier cette recherche de fichiers.
- `Property` : spécifie le nom de la propriété utilisée au moment de l'installation pour tester le résultat de la recherche.

Recherche dans le registre

Les propriétés suivantes permettent de configurer ce type de recherche :

- `Name` : spécifie le nom utilisé dans l'Éditeur des conditions de lancement pour identifier cette recherche dans le registre sélectionné.
- `Property` : spécifie le nom de la propriété utilisée au moment de l'installation pour tester le résultat de la recherche.
- `RegKey` : spécifie une clé du registre à rechercher. Cette valeur doit représenter le chemin complet de la clé du registre. Elle doit correspondre exactement au nom et au chemin d'accès qui figurent dans le registre, espaces éventuels compris, sinon la recherche de la clé dans le registre échoue.
- `Root` : spécifie le point de départ de la recherche dans le Registre. Les valeurs possibles sont : HKLM : débute la recherche dans HKEY_LOCAL_MACHINE, HKCU : débute la recherche dans HKEY_CURRENT_USER, HKCR : débute la recherche dans HKEY_CLASSES_ROOT et HKU : débute la recherche dans HKEY_USERS.
- `Value` : spécifie la valeur à rechercher dans la base de registre. Cette propriété doit être une chaîne correspondant à la valeur affichée dans la colonne Données de l'éditeur du registre Windows.

Recherche de composants

Cette recherche permet de vérifier qu'un composant requis par votre application est disponible sur la machine d'installation. Les propriétés de recherche sont les suivantes :

- `ComponentId` : spécifie le GUID du composant à rechercher. cette valeur doit être fournies sous la forme d'une chaîne mise en forme comme un GUID, à l'aide du format {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX} où X est un chiffre hexadécimal (0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F). Les accolades sont obligatoires.
- `Name` : spécifie le nom utilisé dans l'Éditeur des conditions de lancement pour identifier cette recherche dans le Registre sélectionnée.
- `Property` : spécifie le nom de la propriété utilisée au moment de l'installation pour tester le résultat de la recherche.

Le résultat de ces recherches peut ensuite être utilisé pour ajouter des conditions de lancement de l'installation par le menu contextuel de la rubrique **Conditions de lancement**.

Pour chaque condition de lancement, les propriétés suivantes sont disponibles :

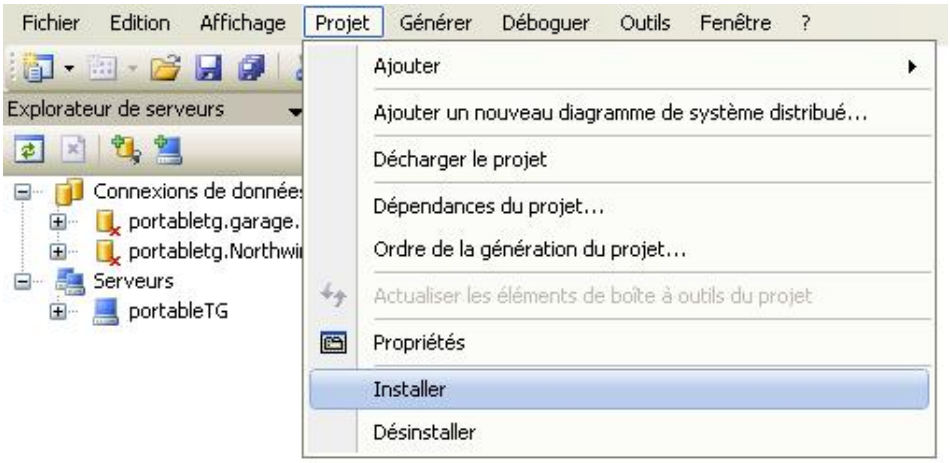
- `InstallUrl` : indique l'emplacement où les fichiers manquant peuvent être téléchargés si une condition de lancement associée à la présence d'un fichier échoue.

- `Message` : spécifie le message à afficher lorsqu'une condition a la valeur false au moment de l'installation.

- `Name` : spécifie le nom utilisé dans l'Éditeur des conditions de lancement pour identifier cette condition de lancement.

- `Condition` : spécifie une condition qui doit être remplie (prendre la valeur true) au moment de l'installation pour que l'installation puisse continuer.

La dernière étape consiste à lancer la génération du projet et à tester son bon fonctionnement par l'intermédiaire de l'option **Install** du menu **Project**.



Déploiement avec ClickOnce

ClickOnce est une technologie de déploiement permettant de créer des applications Windows dont la mise à jour peut être effectuée automatiquement. L'installation de ce type d'application est effectuée avec un minimum d'intervention de la part de l'utilisateur. Cette technique simplifie l'étape de déploiement qui parfois se transforme en véritable casse tête. Les problèmes suivants sont fréquemment rencontrés lors du déploiement d'une application.

Mise à jour de l'application

Avec une méthode de déploiement classique, lorsqu'une nouvelle version de l'application est disponible, l'utilisateur doit en général réinstaller l'application pour bénéficier des mises à jour. La technologie ClickOnce est capable de fournir les mises à jour automatiquement. Dans ce cas seules les parties de l'application qui ont changées sont téléchargées, puis, l'application complète, mise à jour est réinstallée automatiquement à partir d'un nouveau dossier.

Composants partagés

Les applications dépendent souvent de composants partagés, d'où l'existence d'un risque potentiel de conflit de versions. Dans le cas d'un déploiement avec ClickOnce, chaque application est autonome et ne peut pas interférer avec les autres applications.

Autorisations de sécurité

Généralement l'installation d'une application avec une méthode classique exige des autorisations administratives sur le poste de travail où est effectuée l'installation. Le déploiement avec ClickOnce autorise les utilisateurs n'ayant pas de privilèges administratifs à effectuer l'installation et n'attribue que les autorisations de sécurité d'accès du code nécessaires au bon fonctionnement de l'application.

Toutes ces contraintes ont parfois conduit les développeurs à choisir une technologie Web au lieu d'applications Windows classiques simplement pour bénéficier des facilités de déploiement de ce type d'applications. La contre partie de ce choix se retrouve au niveau de la moins bonne réactivité de l'application et d'une interface utilisateur moins élaborée. La technologie ClickOnce rend le déploiement d'applications Windows aussi simple que le déploiement d'applications Web. N'importe quelle application console ou Windows Forms peut être publiée avec ClickOnce. Trois techniques de publication sont disponibles :

- à partir d'une page Web ;
- à partir d'un partage de fichiers réseau ;
- à partir d'un support tel qu'un CD-Rom ou DVD.

L'exécution de l'application dispose également de deux variantes. L'application peut être installée sur l'ordinateur d'un utilisateur et exécutée même si l'ordinateur est hors connexion. Elle peut également être exécutée uniquement en mode en ligne sans installer aucun élément de façon permanente sur l'ordinateur. Les applications ClickOnce sont isolées les unes des autres et l'installation ou l'exécution d'une application ne peut pas interrompre des applications existantes. Par défaut, les applications ClickOnce s'exécutent dans les zones de sécurité Internet ou Intranet. En fonction des besoins l'application peut demander des autorisations de sécurité plus élevées.

La mise à jour de l'application peut également avoir plusieurs modes de fonctionnement. Elles peuvent être automatiques et dans ce cas, l'application vérifie à chaque démarrage si des mises à jour sont disponibles, puis elle les installe automatiquement. L'utilisateur peut manuellement vérifier l'existence d'une mise à jour et décider ou non de son installation. L'administrateur peut aussi rendre obligatoire l'installation d'une mise à jour.

1. Principe de fonctionnement de ClickOnce

Le mécanisme de déploiement ClickOnce repose sur deux fichiers XML appelés manifestes :

- Un manifeste d'application.
- Un manifeste de déploiement.

Le manifeste d'application décrit l'application elle-même, les assemblies et les fichiers qui la composent, les dépendances, les autorisations requises pour l'exécution et l'emplacement dans lequel les mises à jour seront disponibles.

Le manifeste de déploiement décrit comment l'application est déployée, y compris l'emplacement du manifeste d'application et la version de l'application que les clients doivent exécuter. Ces deux fichiers sont générés par Visual Studio.

Le manifeste de déploiement est copié vers l'emplacement de déploiement. Cet emplacement peut être un serveur Web, un répertoire partagé sur le réseau ou des supports amovibles tels qu'un CD-Rom. Le manifeste d'application et tous les fichiers de l'application sont également copiés vers un l'emplacement de déploiement spécifié dans le manifeste de déploiement. Ces fichiers peuvent être copiés vers le même emplacement ou dans deux emplacements distincts. Visual Studio prend également en charge les copies de ces fichiers.

Après le déploiement de l'application dans l'emplacement de déploiement, les utilisateurs peuvent télécharger et installer l'application en cliquant sur l'icône représentant le fichier manifeste de déploiement disponible sur une page Web ou dans un dossier. L'utilisateur voit simplement s'afficher une simple boîte de dialogue lui demandant de confirmer l'installation. Après validation, l'installation continue et l'application est lancée sans autre intervention. Si l'application nécessite des autorisations d'exécution plus élevées, la boîte de dialogue demande à l'utilisateur d'accorder les autorisations pour que l'installation puisse se poursuivre.

L'application est ajoutée au menu **Démarrer** de l'utilisateur et à la rubrique **Ajout/Suppression de programmes** du **Panneau de configuration**. Contrairement à d'autres technologies de déploiement, rien n'est ajouté au dossier **Program Files**, dans la base de registre ou sur le bureau. De plus aucun droit d'administration n'est nécessaire pour l'installation.

Lorsque vous créez une version mise à jour de l'application, vous devez également générer un nouveau manifeste d'application et copier les fichiers vers un emplacement de déploiement, généralement un dossier frère du dossier de déploiement d'origine. Le manifeste doit aussi être mis à jour pour qu'il pointe vers l'emplacement de la nouvelle version de l'application.

2. Les différentes méthodes de déploiement

Pour déployer une application ClickOnce, trois stratégies sont possibles. La stratégie que vous choisissez dépend principalement du type d'application que vous déployez. Les trois stratégies de déploiement sont les suivantes :

- Installation à partir du Web ou d'un partage réseau ;
- Installation à partir d'un CD-Rom ;
- Démarrage de l'application à partir du Web ou d'un partage réseau.

Installation à partir du Web ou d'un partage réseau

Cette stratégie permet de déployer votre application sur un serveur Web ou un partage de fichiers réseau. Lorsqu'un utilisateur final souhaite installer l'application, il clique sur une icône d'une page Web ou double clique sur une icône du partage de fichiers. L'application est ensuite téléchargée, installée et démarrée sur l'ordinateur de l'utilisateur. Des éléments sont ajoutés au menu **Démarrer** et au groupe **Ajout/Suppression de programmes** dans le **Panneau de configuration**.

Étant donné que cette stratégie dépend de la connexion réseau, elle fonctionne de manière optimale pour les applications qui seront déployées pour les utilisateurs qui ont accès à un réseau local ou une connexion Internet rapide.

Installation à partir d'un CD-Rom

Cette stratégie permet de déployer votre application sur un support amovible tel qu'un CD-Rom ou un DVD. Comme pour l'option précédente, lorsque l'utilisateur choisit d'installer l'application, cette dernière est installée, lancée et des éléments sont ajoutés au menu **Démarrer** et au groupe **Ajout/Suppression de programmes** dans le **Panneau de configuration**.

Cette stratégie fonctionne mieux dans le cas d'applications déployées sur les ordinateurs d'utilisateurs qui ne possèdent pas une connectivité réseau persistante ou qui ont des connexions à faible bande passante. L'application étant installée à partir d'un support amovible, aucune connexion réseau n'est nécessaire pour l'installation ; la connectivité réseau est néanmoins nécessaire pour les mises à jour de l'application.

Démarrage de l'application à partir du Web ou d'un partage réseau

Cette stratégie est similaire à la première, sauf que l'application se comporte comme une application Web. Lorsque l'utilisateur clique sur un lien d'une page Web (ou double clic sur une icône du partage de fichiers), l'application est lancée. Lorsque les utilisateurs ferment l'application, cette dernière n'est plus disponible sur leur ordinateur local. Aucun élément n'est ajouté au menu **Démarrer** ou au groupe **Ajout/Suppression de programmes** dans le **Panneau de configuration**. Techniquement, l'application est téléchargée et installée dans un cache d'application de l'ordinateur local, de la même façon qu'une application Web est téléchargée vers le cache Web. Comme pour le cache Web, les

fichiers sont nettoyés du cache d'application en fin d'utilisation. Toutefois, l'utilisateur a l'impression que l'application est exécutée à partir du Web ou du partage de fichiers.

Cette stratégie est à privilégier pour les applications rarement utilisées.

3. Les mises à jour de l'application

ClickOnce peut fournir automatiquement les mises à jour de l'application. Une application ClickOnce lit périodiquement son fichier manifeste de déploiement pour vérifier si les mises à jour de l'application sont disponibles. Si elle est disponible, la nouvelle version de l'application est téléchargée et exécutée. Pour des raisons d'efficacité, seuls les fichiers modifiés sont téléchargés.

Trois stratégies de base sont possibles pour les mises à jour :

- La vérification des mises à jour au démarrage de l'application ;
- La vérification des mises à jour après le démarrage de l'application (exécutée dans un thread d'arrière-plan) ;
- La présentation d'une interface utilisateur destinée aux mises à jour.

Vous pouvez également déterminer la fréquence de vérification des mises à jour effectuée par l'application ou configurer une mise à jour obligatoire. Les mises à jour d'application exigent une connexion au réseau. En l'absence d'une connexion réseau, l'application s'exécute sans vérifier les mises à jour, quelle que soit la stratégie de mise à jour choisie.

Vérification des mises à jour après le démarrage

Par défaut, l'application tente de localiser et de lire le fichier manifeste de déploiement en arrière-plan pendant son exécution. Si une mise à jour est disponible, lors de la prochaine exécution, l'utilisateur sera invité à télécharger et à installer la mise à jour.

Cette stratégie est tout particulièrement adaptée aux connexions réseau à bande passante restreinte ou aux applications volumineuses, pouvant nécessiter de longs téléchargements.

Vérification des mises à jour au démarrage

Avec cette stratégie, l'application tente de localiser et de lire le fichier manifeste de déploiement à chaque lancement. Si une mise à jour est disponible, elle sera téléchargée et exécutée ; sinon, la version existante de l'application sera exécutée.

Cette stratégie est bien adaptée aux connexions réseau à large bande passante ; le délai nécessaire au lancement de l'application peut être inacceptable sur des connexions à bande passante plus restreinte du fait du téléchargement des mises à jour.

Mises à jour obligatoire

Il est parfois souhaitable d'obliger les utilisateurs à exécuter une version mise à jour de l'application si, par exemple, vous avez modifié une ressource qui risque de perturber le fonctionnement de l'ancienne version de l'application. Vous pouvez dans ce cas marquer la mise à jour comme étant obligatoire et de ce fait, empêcher l'exécution d'une version plus ancienne de l'application. Cette stratégie doit être associée avec la vérification des mises à jour au démarrage.

Intervalles de mise à jour

Dans le cadre des mises à jour automatiques vous pouvez également spécifier la fréquence de vérification des mises à jour. Par exemple, vous pouvez souhaiter une vérification à chaque exécution de l'application, une fois par semaine ou une fois par mois. Si aucune connexion réseau n'est disponible au moment spécifié pour la vérification, celle-ci est effectuée à la prochaine exécution de l'application.

Blocage des mises à jour

Il est également possible de faire en sorte que votre application ne vérifie jamais les mises à jour. Par exemple, vous pouvez déployer une application simple qui ne sera jamais mise à jour tout en bénéficiant de la facilité d'installation fournie par ClickOnce.

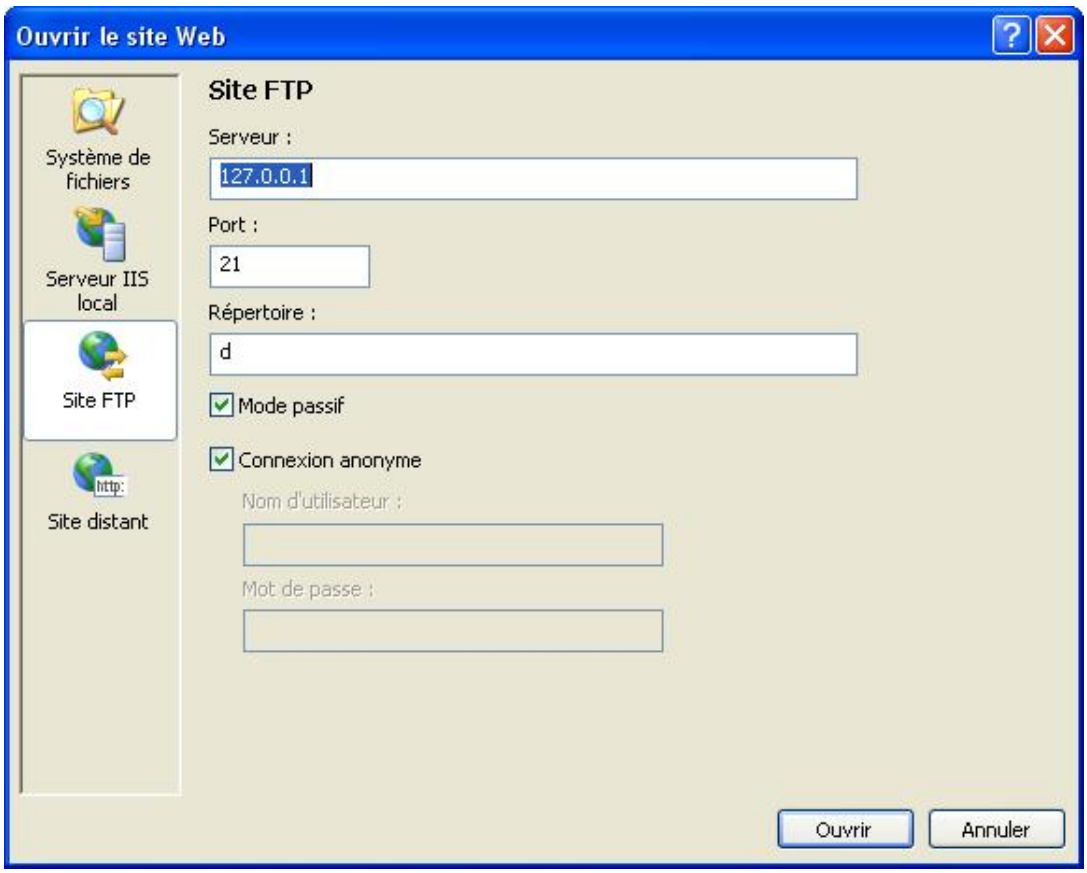
4. Mise en œuvre de la publication ClickOnce

La publication d'une application avec la technologie ClickOnce est grandement facilitée par un assistant permettant de recueillir la majorité des informations nécessaires au déploiement. Cet assistant est disponible en choisissant l'option **Publier** du menu contextuel accessible sur le projet à déployer dans l'explorateur de solutions. Certaines options de déploiement ne sont cependant pas gérées par cet assistant et doivent être configurées manuellement via la boîte de dialogue de propriétés du projet.

La première étape de l'assistant consiste à configurer l'emplacement où doit se faire la publication.

Cet emplacement peut être :

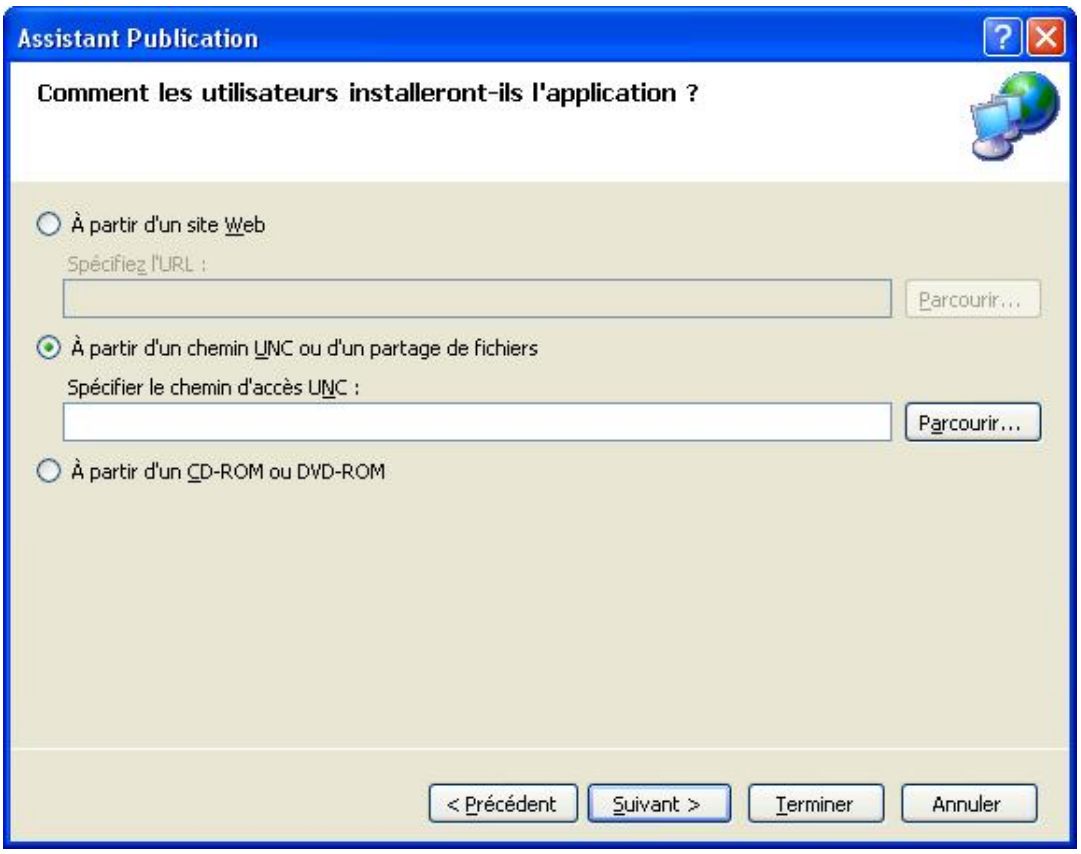
- Un répertoire de la machine.
- Un répertoire partagé sur une autre machine en indiquant un chemin UNC de la forme suivante \\nom de la machine\nom du répertoire. Vous devez avoir l'autorisation d'écrire sur le partage pour que la publication puisse être réalisée.
- Le serveur Web IIS de la machine sur lequel vous devez avoir au préalable ajouté un répertoire virtuel pour accueillir les fichiers.
- Un serveur ftp pour lequel vous devez fournir les informations de connexion à l'aide de la boîte de dialogue suivante :



Vous devez indiquer :

- L'adresse IP ou le nom du serveur ftp.
- Le numéro du port utilisé pour contacter le serveur (en général 21).
- Le répertoire du serveur dans lequel sera effectuée la copie des fichiers. Vous devez avoir l'autorisation d'écriture dans ce répertoire.
- Si vous êtes situé derrière un pare-feu en activant l'option **Mode passif**.
- Si vous vous connectez de façon anonyme ou sinon le nom d'utilisateur et le mot de passe utilisés pour la connexion.

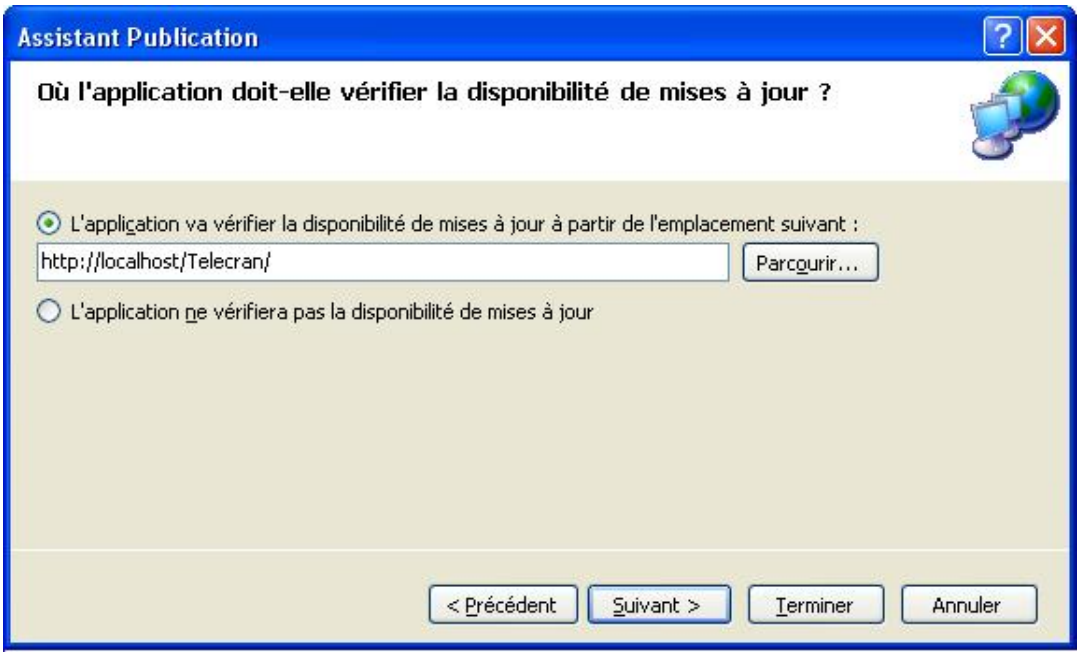
La deuxième étape détermine comment les utilisateurs vont installer l'application.



Les options possibles sont :

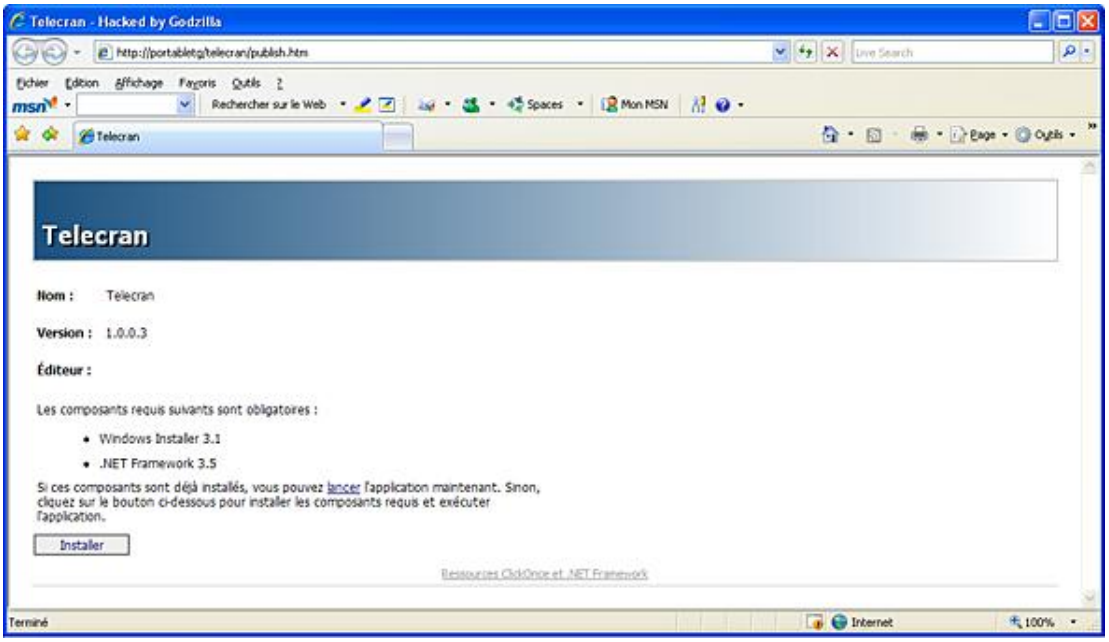
- À partir d'un site Web dont vous indiquez l'URL.
- À partir d'un partage réseau dont vous spécifiez le chemin UNC. Les utilisateurs devront bien sûr avoir le droit de lire sur le partage. Le droit d'écriture n'est pas obligatoire et même fortement déconseillé.
- À partir d'un CD-Rom ou DVD que vous fournirez. La création de ce support n'est pas réalisé par l'assistant et doit être effectué par une application de gravure externe.

L'assistant vous propose ensuite de configurer la stratégie de mise à jour.

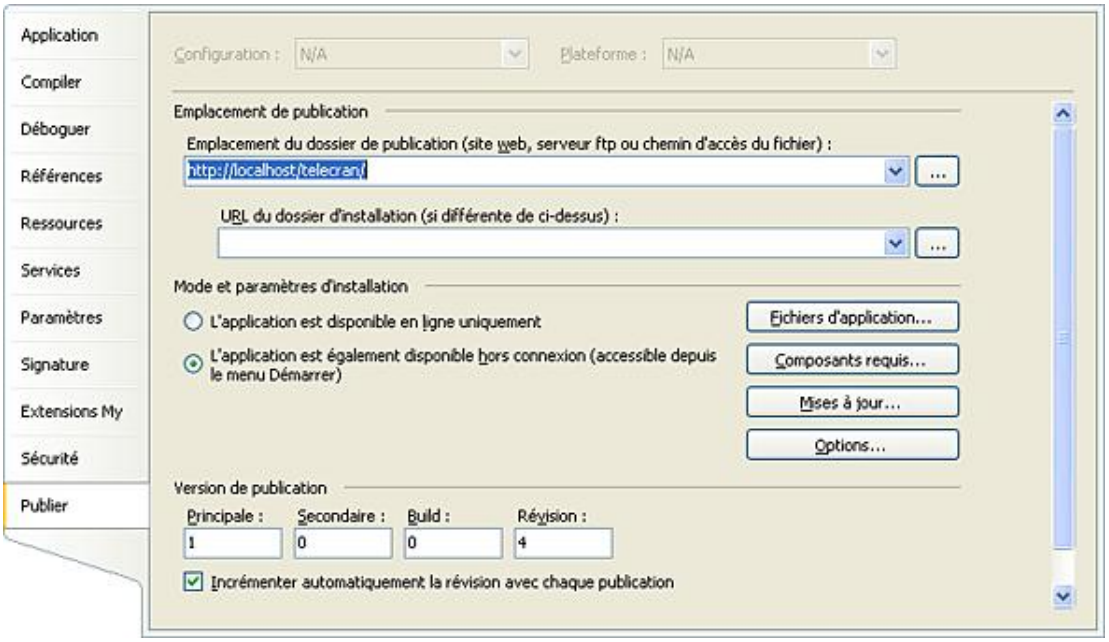


Si vous souhaitez activer les mises à jour, il faut indiquer l'emplacement à partir duquel elles seront obtenues. Cet emplacement n'est pas forcément le même que celui utilisé pour l'installation. L'ultime étape affiche un résumé des informations sélectionnées et permet de lancer la publication avec le bouton **Finish**.

À la fin de l'installation une page html est ouverte sur l'emplacement utilisé pendant la publication permettant le lancement de l'installation ou l'exécution de l'application.

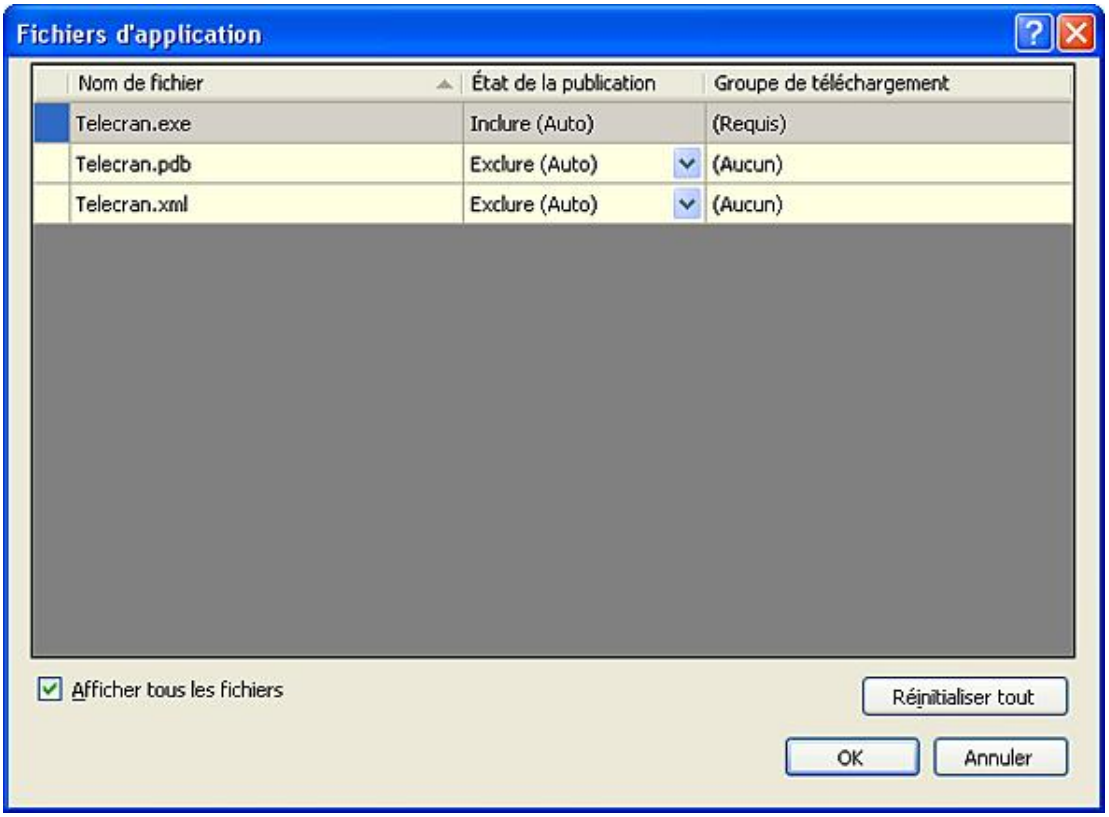


Les options de déploiement plus spécifiques doivent être configurées via la rubrique **Publier** des propriétés du projet. Cette boîte de dialogue reprend les propriétés configurées par l'assistant de publication.



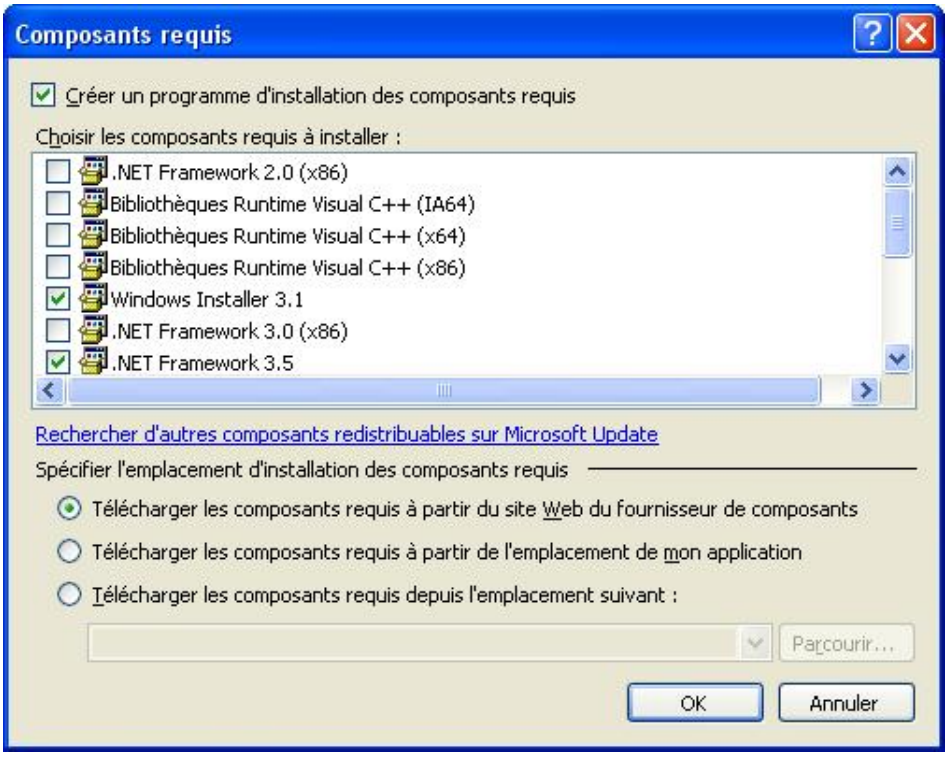
Les boutons **Fichiers d'application**, **Composants requis**, **Mises à jour** et **Options** permettent de peaufiner les réglages.

Le bouton **Fichiers d'application** affiche la boîte de dialogue suivante concernant les fichiers constituant l'application.



L'état de la publication de chaque fichier peut être configuré avec trois valeurs différentes :

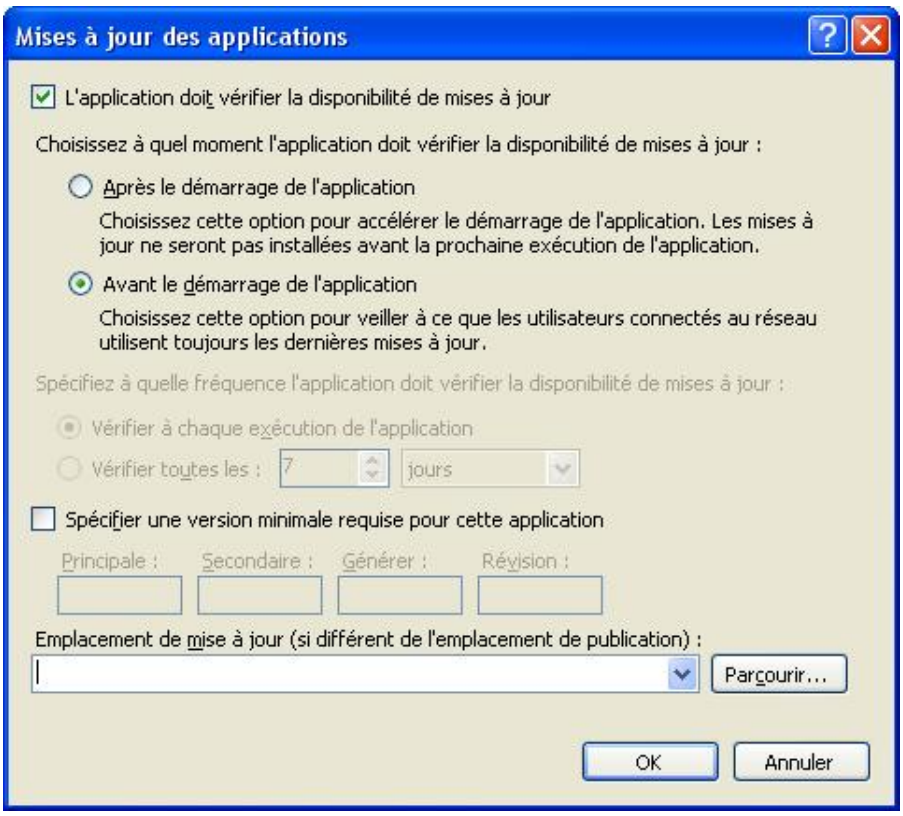
- **Inclure** : le fichier sera disponible pour les utilisateurs sur le support de déploiement.
- **Exclure** : le fichier n'est pas recopié sur le support de déploiement.
- **Fichier de données** : le fichier contient des données nécessaires au bon fonctionnement de l'application et sera inclus dans la publication. Le bouton **Composants requis**est utilisé pour configurer les éléments nécessaires pour le fonctionnement de l'application.



Vous pouvez choisir de créer un programme d'installation pour les composants requis pour le fonctionnement de l'application en cochant la case **Créer un programme d'installation des composants requis**. Les composants concernés sont à choisir dans la liste présentée. Vous devez également indiquer à partir de quel emplacement ces composants seront installés. Trois options sont possibles :

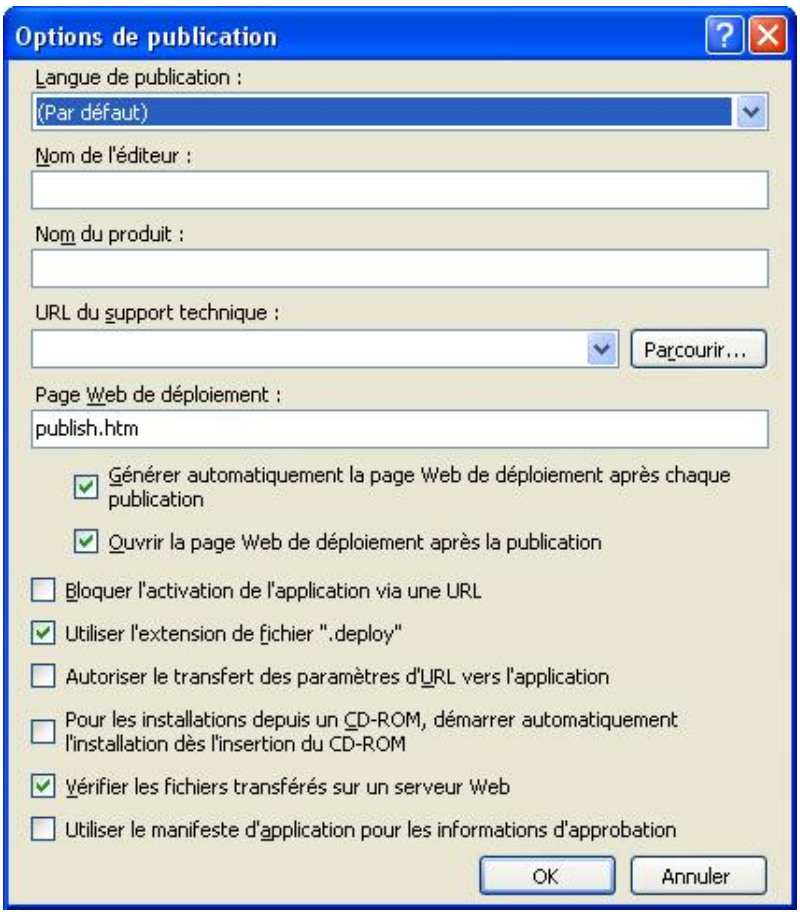
- à partir du site Web du fournisseur du composant ;
- à partir du même emplacement que celui utilisé pour installer l'application ;
- à partir de l'emplacement indiqué.

La configuration des mises à jour prévue lors de l'utilisation de l'assistant peut être modifiée par le bouton **Mises à jour**.



La case à cocher **L'application doit vérifier la disponibilité de mises à jour**spécifie que l'application doit vérifier la disponibilité de mises à jour dès son installation. Si vous sélectionnez cette option, les autres options deviennent disponibles. Elles permettent de choisir le moment où aura lieu la vérification de la disponibilité d'une mise à jour. L'option **Avant le démarrage de l'application** indique que l'application doit vérifie la disponibilité de mises à jour avant le démarrage. Cela garantit que les utilisateurs qui sont connectés au réseau disposent toujours de la version la plus récente de l'application. Cette option peut ralentir le démarrage de l'application dans le cas où des mises à jour sont disponibles. L'option **Après le démarrage de l'application** planifie l'exécution de la mise à jour lors du prochain redémarrage de l'application. La fréquence de vérification des mises à jour peut également être indiquée par un nombre d'heures, de jours ou de semaines ou bien être exécutée à chaque démarrage de l'application. Vous devez aussi indiquer l'emplacement à partir duquel les mises à jour sont disponibles si celui-ci est différent de l'emplacement d'installation.

Le dernier bouton va servir à configurer diverses options de déploiement.



Les options suivantes sont disponibles :

Langue de publication

Spécifie la langue (et les paramètres régionaux) dans laquelle l'application doit être publiée.

Nom de l'éditeur

Spécifie le nom de l'éditeur de l'application. Si cette zone est vide, la valeur de la propriété `RegisteredOrganization` de l'ordinateur sera utilisée. Si cette valeur est nulle, le nom du projet est utilisé.

Nom du produit

Spécifie le nom de produit de l'application. Si le nom de produit est vide, le nom de l'assembly est utilisé.

URL du support technique

Spécifie un site Web qui contient des informations d'assistance pour votre application. La spécification de cette URL est facultative. Si elle est utilisée, cette URL apparaît dans l'entrée **Ajout/Suppression de programmes** pour votre application dans le **Panneau de configuration** de Windows.

Page Web de déploiement

Spécifie un nom pour la page Web de déploiement. Le nom de fichier par défaut est Publish.htm.

Générer automatiquement la page Web de déploiement après chaque publication

Si cette option est sélectionnée, le processus de publication génère une page Web de déploiement à chaque publication. Cette option n'est disponible que si une Page Web de déploiement est spécifiée.

Ouvrir la page Web de déploiement après la publication

Si cette option est sélectionnée, la page Web de déploiement générée automatiquement s'ouvre après la publication.

Bloquer l'activation de l'application via une URL

Si cette option est désactivée, l'application s'exécute automatiquement après l'installation. Si elle est activée, l'utilisateur devra démarrer l'application à partir du raccourci de programme dans le Menu **Démarrer**.

Utiliser l'extension de fichier ".deploy"

Si cette option est sélectionnée, le fichier de déploiement utilise l'extension .deploy. Certains serveurs Web sont configurés pour bloquer, par raison de sécurité, les fichiers qui ne sont pas habituellement présents dans un contenu Web. Par exemple, les fichiers portant les extensions suivantes peuvent être bloqués : .dll, .config, .mdf. Les applications Windows contiennent généralement des fichiers portant certaines de ces extensions. Si un utilisateur essaie d'exécuter une application ClickOnce qui accède à un fichier bloqué sur un serveur Web, une erreur se produit. Plutôt que de débloquer toutes les extensions de fichier, chaque fichier d'application est publié par défaut avec une extension de fichier ".deploy". Si cette option est utilisée, le serveur Web ne doit être configuré que pour débloquer les trois extensions de fichier suivantes :

- .application
- .manifest
- .deploy

Autoriser le transfert des paramètres d'URL vers l'application

Par défaut cette option est désactivée. Si cette option est activée, l'application sera capable d'accéder et de traiter les informations de paramètre de l'URL.

Pour les installations depuis un CD-ROM, démarrer automatiquement l'installation dès l'insertion du CD-ROM

Si cette option est sélectionnée, elle ajoute un fichier Autorun.inf à la racine du support pour les applications ClickOnce qui sont installées via CD-Rom ou DVD-Rom.

Vérifier les fichiers transférés sur un serveur Web

Si cette option est activée, le processus de publication télécharge chaque fichier pour vérifier qu'ils peuvent bien être téléchargés. Vous êtes informés des fichiers qui ne peuvent pas être téléchargés.

Utiliser le manifeste d'application pour les informations d'approbation

Lorsque cette option est sélectionnée, vous pouvez signer à nouveau le manifeste de l'application à l'aide d'un certificat contenant vos propres coordonnées.