collection Solutions informatiques

C# 4

Développez des applications Windows avec Visual Studio 2010

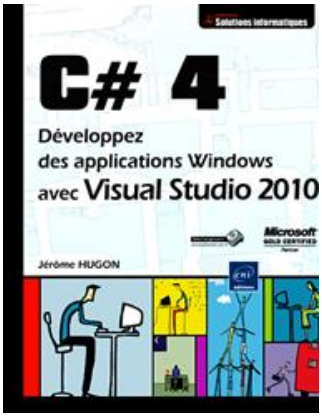
Microsoft GOLD CERTIFIED Partner

Jérôme HUGON

C# 4

Développez des applications Windows avec Visual Studio 2010

Jérôme HUGON



Résumé

Ce livre sur le développement d'applications Windows avec le langage C# et Visual Studio 2010 est destiné aux **développeurs qui débutent avec le framework .NET**. Il leur permet d'apprendre les **bases du langage C#** et introduit des **concepts plus avancés** leur donnant une vue d'ensemble des possibilités offertes par le langage C#, Visual Studio et le framework .NET en général. L'auteur a choisi une approche pas à pas tout en construisant une **application fonctionnelle tout au long de l'ouvrage** pour illustrer de manière pratique et cohérente les concepts abordés.
L'apprentissage commence par la familiarisation avec **l'interface de Visual Studio 2010** ainsi qu'avec **le concept de l'architecture .NET**. Les détails du langage C# (en version 4.0 au moment de la rédaction du livre), sa **syntaxe** et ses fonctionnalités comme les **classes**, **l'héritage**, les **interfaces**, les **types génériques** ou encore les **délégués** et les **évènements** sont ensuite expliqués avant d'aborder la conception d'interfaces utilisateur.
La **conception de l'interface utilisateur** couvre toutes les phases utiles pour créer des applications Windows à la fois fonctionnelles et ergonomiques, allant de la **création de formulaires** à la **création de contrôles** en passant par l'implémentation de **gestionnaire d'évènements** et la validation des données saisies. Une introduction à la conception de **formulaires WPF** (Windows Presentation Foundation) est également incluse.
Les outils de Visual Studio qui permettent de réaliser **les tests et le débogage** des applications sont également détaillés en présentant les techniques de **gestion des erreurs** mais aussi les concepts permettant de surveiller les applications comme le **traçage**, l'interaction avec **les journaux d'évènements** et l'utilisation **des compteurs de performance**.
L'utilisation de **Entity Framework 4** est détaillée au sein d'exemples concrets permettant de comprendre rapidement comment créer des **modèles de données** et comment les utiliser pour communiquer avec une base de données tout en apprenant à utiliser le **langage de requête LINQ** pour interagir avec des données sous différents formats (objets, SQL ou XML). L'alternative du stockage de données d'une application sur le système de fichiers et l'utilisation du **concept de la sérialisation** sont également détaillés fournissant ainsi une vision globale des possibilités offertes par le framework .NET concernant la gestion des données.
Des **concepts plus avancés** sont également abordés afin d'exposer une gamme plus large des possibilités offertes par le langage C# et Visual Studio : l'utilisation des **expressions régulières**, le développement **d'applications multi-tâches**, la **globalisation** et la **localisation** d'une application, la **sécurité** du code, l'implémentation d'applications client/serveur, le dessin avec **GDI+** ainsi que la **réflexion** font partie des sujets introduits.
La dernière partie de l'ouvrage est consacrée à la **création d'assemblages** ainsi qu'au **déploiement des applications**. Les outils et techniques mis à disposition par Visual Studio pour créer des **installeurs Windows et configurer les applications** y sont détaillés.
Le code de l'application exemple traitée dans l'ouvrage est en téléchargement sur le site www.editions-eni.fr.

Les chapitres du livre :

L'auteur

Jérôme HUGON est développeur / consultant .NET depuis de nombreuses années pour LABEL. THE brand.intelligence™ COMPANY. Il est certifié Microsoft sur les technologies .NET. Son expérience du développement de sites web ASP.NET et SharePoint comme d'applications Windows ou Microsoft Surface lui permet d'apporter au lecteur une bonne connaissance de base du langage C# mais aussi une belle ouverture vers des concepts plus avancés pour une utilisation optimale de ce langage.

Avant-propos

Cet ouvrage est destiné aux développeurs débutants ou plus expérimentés non familiarisés avec les outils de développement de Microsoft et qui souhaitent pouvoir utiliser de la meilleure manière possible les capacités offertes par le Framework .NET au travers du langage C#.

Le langage C#, associé au Framework .NET, permet de développer de nombreux types d'applications. Il peut s'agir d'applications Windows comme au fil de cet ouvrage, d'applications Web, de services Windows, d'applications WPF, Silverlight ou Microsoft Surface.

Visual Studio est l'outil, fourni par Microsoft, indispensable pour travailler le plus efficacement possible avec le Framework .NET et le langage C#. La première partie de l'ouvrage vous aidera à vous familiariser avec l'environnement de développement et les nombreuses possibilités offertes par Visual Studio 2010 pour mener à bien toutes les étapes de la création d'une application.

Au fil de l'ouvrage, vous serez amené à construire une application complète et chacune des grandes étapes du cycle de développement y sera abordée en commençant par la définition des classes métier. Vous aborderez les détails du langage C# tant dans les concepts de base que dans les techniques plus avancées.

La conception d'applications Windows repose en grande partie sur l'interface utilisateur. C'est grâce à elle que l'utilisateur interagit avec l'application et inversement. Les outils de Visual Studio sont décrits au travers d'exemples concrets permettant d'apprendre la conception de formulaires et de contrôles utilisateurs. Les processus de validation des données et l'ergonomie sont également des points abordés.

Il n'existe pas d'applications sans bug, seulement des applications dont les bugs ne sont pas encore identifiés. Ce n'est pas toujours l'application qui crée l'erreur, cela peut aussi être le système sur lequel elle sera déployée et qui n'existe pas encore au moment du développement. C'est pour ces raisons que l'ouvrage vous fera découvrir les puissants outils de Visual Studio permettant de tester, débugger et analyser une application pendant la phase de conception et après son déploiement.

Les différentes formes d'enregistrement des données seront présentées en mettant l'accent sur les bases de données et l'Entity Framework en tant que couche d'accès aux données. Vous apprendrez à créer une base de données à partir de la définition de classes C# et inversement, c'est-à-dire créer la définition de classes à partir d'un schéma de base de données. Vous utiliserez le langage de requêtes LINQ pour accéder aux données, les ajouter, les mettre à jour ou les supprimer. L'enregistrement des données sur le système de fichiers est également une technique répandue pour les applications Windows, c'est pourquoi les possibilités d'interactions avec le système de fichiers sont décrites dans cet ouvrage ainsi que la mise en pratique de la sérialisation, concept fourni par le Framework .NET, permettant de sauvegarder et restaurer facilement des objets sous forme de fichiers binaires ou XML.

La globalisation et la localisation d'une application sont des concepts primordiaux qui sont décrits au travers d'exemples concrets pour vous aider à appréhender au mieux la création d'une application internationale. Vous pourrez également vous familiariser avec les techniques plus avancées comme les expressions régulières, le développement multitâche ou encore l'implémentation de la sécurité dans vos applications. Pour finir, une introduction aux concepts de Reflection, de Remoting et de dessin avec GDI+ vous est proposée.

Comme pour la création d'une application, cet ouvrage se clôturera par l'apprentissage du déploiement d'applications. Vous apprendrez à créer des assemblages et des programmes d'installation grâce à la description de toutes les étapes et de toutes les possibilités de configuration de celles-ci.

Introduction

Visual Studio est l'interface de développement de Microsoft. Elle est composée d'un ensemble d'outils permettant aux développeurs de créer des applications pour les plateformes .NET. Visual Studio 2010 est distribué en plusieurs éditions :

- *Express* : Microsoft fournit gratuitement cette édition limitée de Visual Studio 2010 dans un but de formation pour les développeurs. Elle réunit toutes les fonctionnalités de base pour la création de projets.

- *Professional* : édition à destination des développeurs professionnels seuls ou au sein de petites équipes. Les outils de débogage et de tests unitaires font partie des fonctionnalités notables de cette édition.

- *Premium* : pour les équipes professionnelles travaillant sur des projets nécessitant plus d'interactions entre leurs membres. En plus des fonctionnalités de l'édition Professional, cette édition offre les fonctionnalités de test de l'UI (Interface utilisateur), de développement de base de données et de la génération au déploiement en passant par les tests.

- *Ultimate* : en plus des fonctionnalités de l'édition Premium, cette édition contient l'IntelliTrace, outil supplémentaire pour le débogage d'applications, le gestionnaire de tests, et des outils de design d'architecture d'applications.

Dans cet ouvrage, l'édition Professional sera utilisée pour la présentation des exemples bien que la quasi-totalité soit valable avec l'édition Express.

L'édition Express de Visual C# 2010 Express est disponible au téléchargement : http://www.microsoft.com/express/downloads/.

L’interface de développement

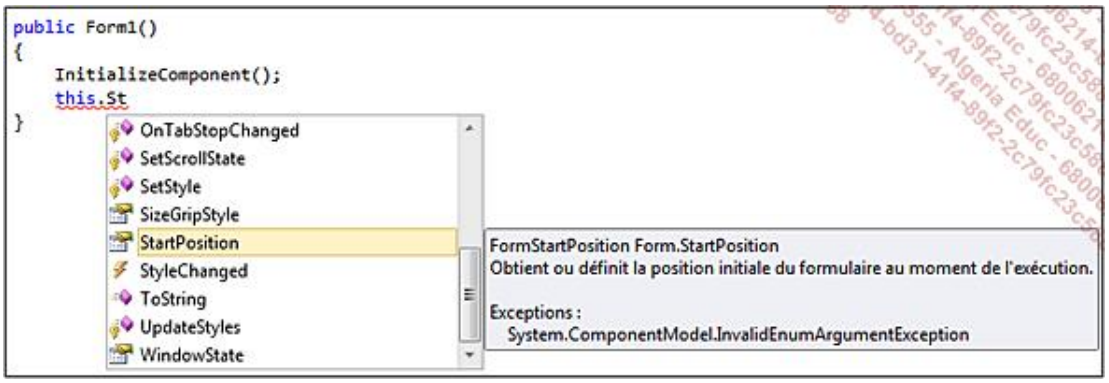
De nombreuses nouveautés ont été apportées à cette version de Visual Studio par rapport aux précédentes. L’éditeur a entièrement été refait pour se baser sur le Framework .NET 4. Les applications produites peuvent spécifier quelle version cible du Framework .NET utiliser (incluant les versions 2.0, 3.0, 3.5 et 4), l’IntelliSense est désormais supporté pour le JavaScript.

L’interface de Visual Studio comporte plusieurs outils indispensables au développement d’applications.

1. L’éditeur de texte

L’éditeur de texte de Visual Studio est un puissant outil permettant de saisir le code de l’application. Les mots clés, les types y sont colorés pour faciliter la lecture et la compréhension du code. En même temps que le code est saisi, l’éditeur évalue les erreurs de syntaxe, les variables déclarées mais non utilisées dans le code et affiche l’IntelliSense.

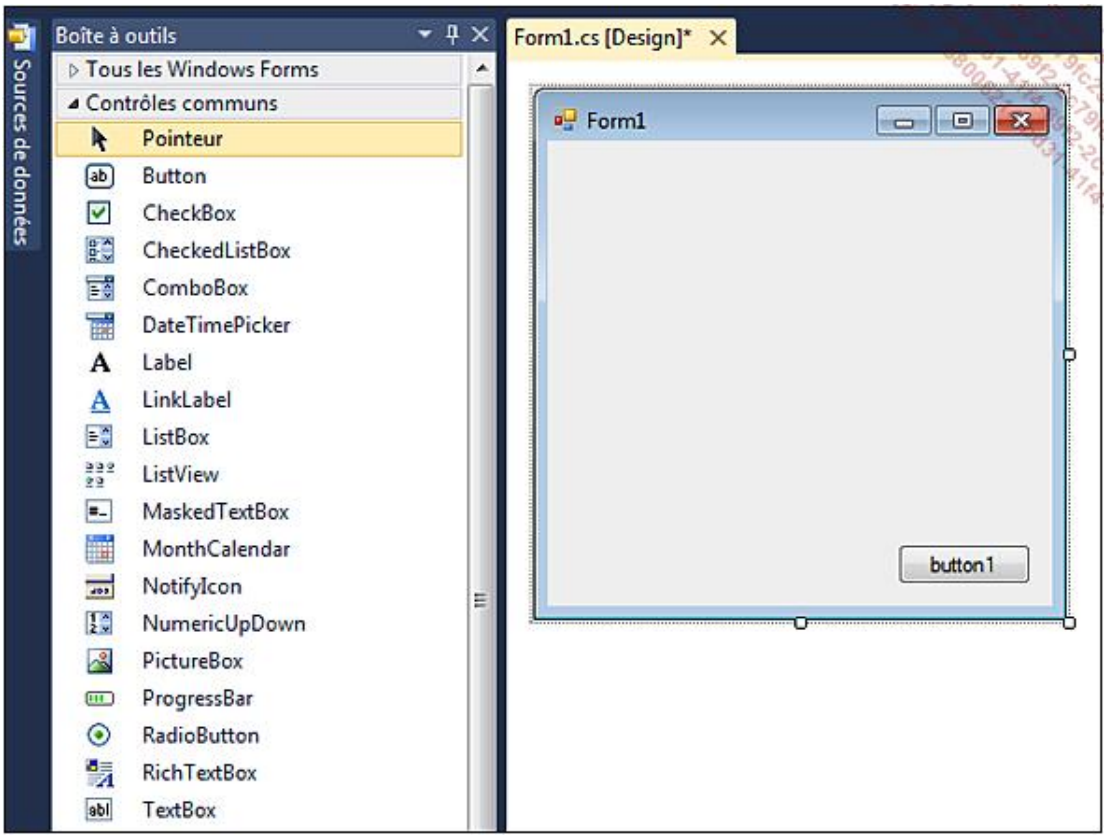
L’IntelliSense est une fonctionnalité permettant d’afficher les classes et leurs membres en rapport avec le code saisi ainsi que les paramètres et les surcharges possibles pour les méthodes :



Les icônes à gauche des noms de membres permettent de définir visuellement si le membre est une méthode, une propriété, une variable, un évènement ou un autre type de membre. Lorsqu’un membre est en surbrillance, une info bulle explicative affiche une description de la méthode, les paramètres attendus ou encore les types d’exceptions qui peuvent être levés.

2. Le concepteur de vues

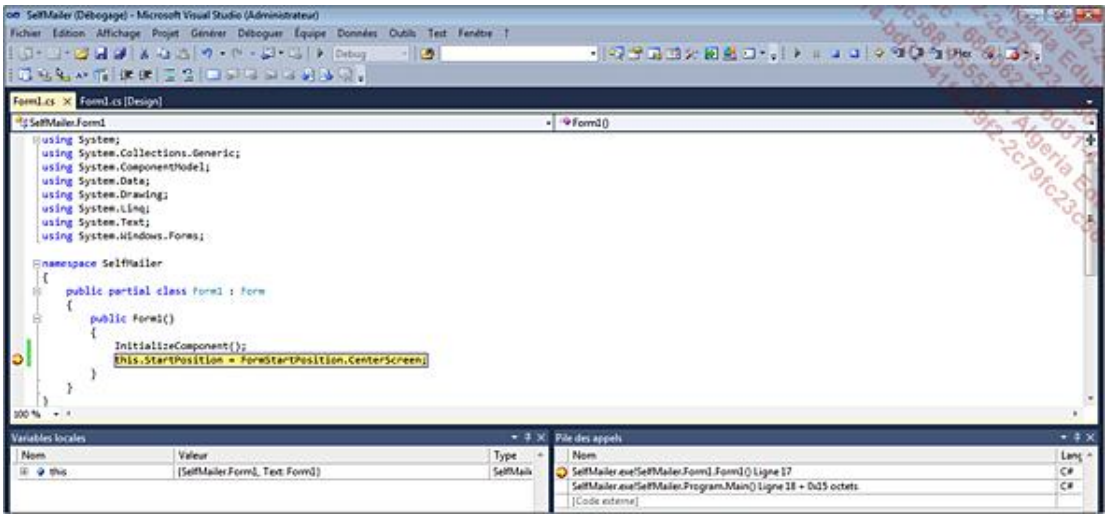
Le concepteur de vue permet de placer graphiquement les éléments sur les formulaires en les faisant glisser depuis la boîte à outils :



Visual Studio génère automatiquement le code requis pour instancier les contrôles et les positionner. Le concepteur de vue est en quelque sorte un outil qui interprète le code pour en donner une représentation visuelle.

3. Le débuggeur intégré

Le débuggeur intégré de Visual Studio permet de tester et suivre le déroulement de l'application, de disposer des points d'arrêts et de suivre pas à pas l'exécution du code, de surveiller et de modifier manuellement les variables en cours d'exécution :

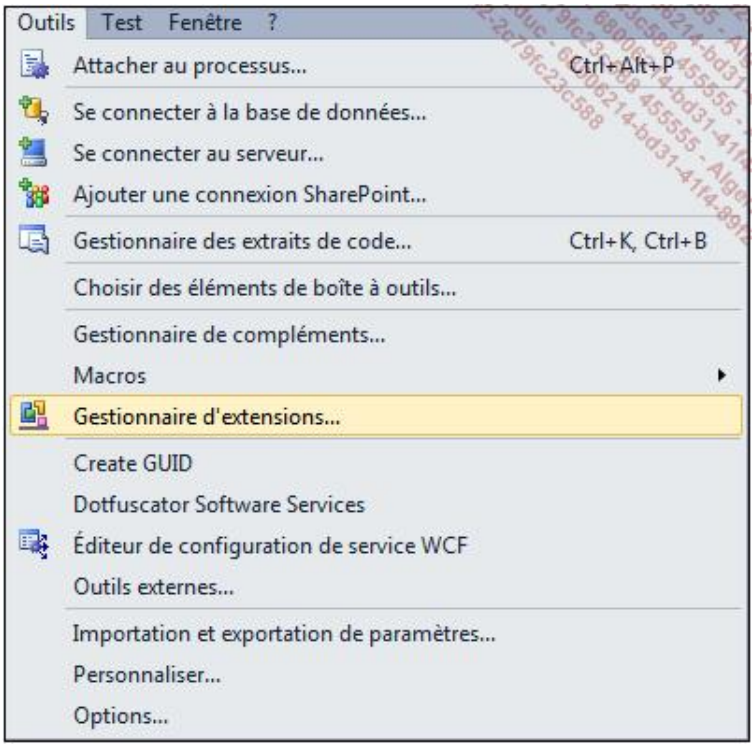


Lorsque l'application est lancée depuis Visual Studio avec la touche **F5** (**Démarrer le débogage**), Visual Studio attache son débuggeur au processus de l'application et permet de la contrôler.

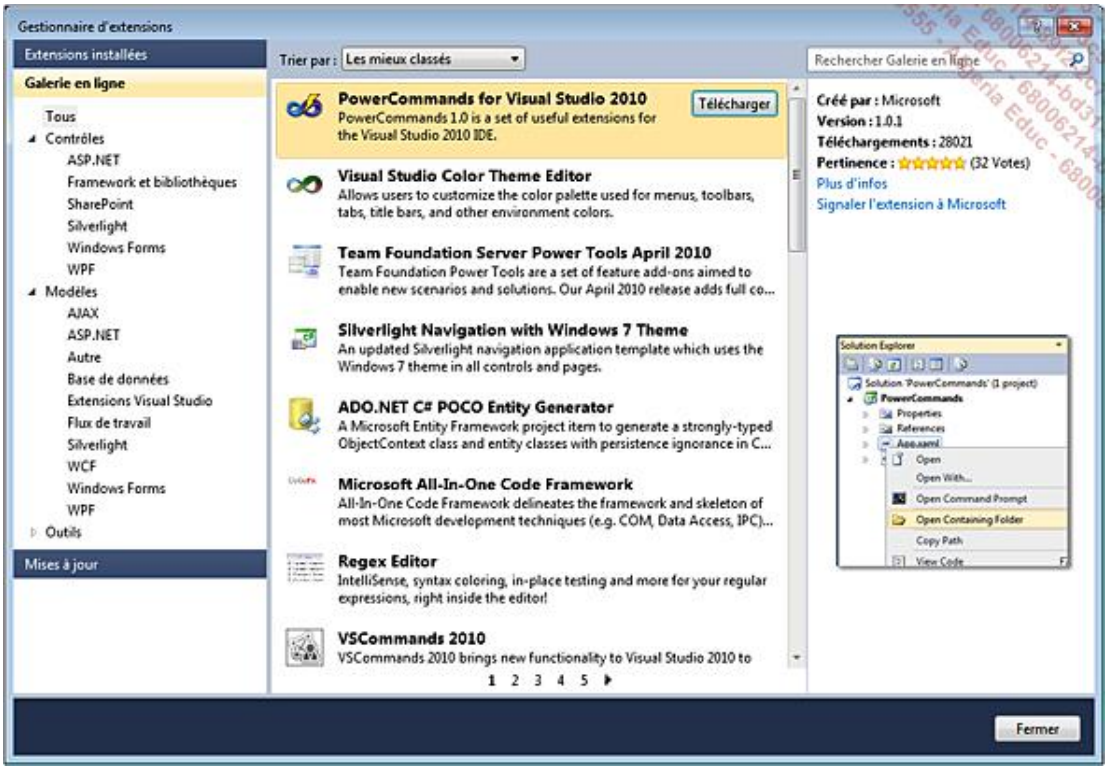
Le débuggeur et ses fonctionnalités seront détaillés dans le chapitre Débogage.

4. Le gestionnaire d'extensions

Le gestionnaire d'extensions est une nouveauté de la version 2010 de Visual Studio. Il permet d'ajouter des fonctionnalités à Visual Studio à partir d'une galerie en ligne. Le gestionnaire d'extensions est accessible depuis le menu **Outils** puis **Gestionnaire d'extensions...** :



L'onglet **Galerie en ligne** propose une multitude d'extensions. Que ce soit des contrôles, des modèles ou des outils, il suffit de parcourir la galerie et de cliquer sur le bouton **Télécharger** :



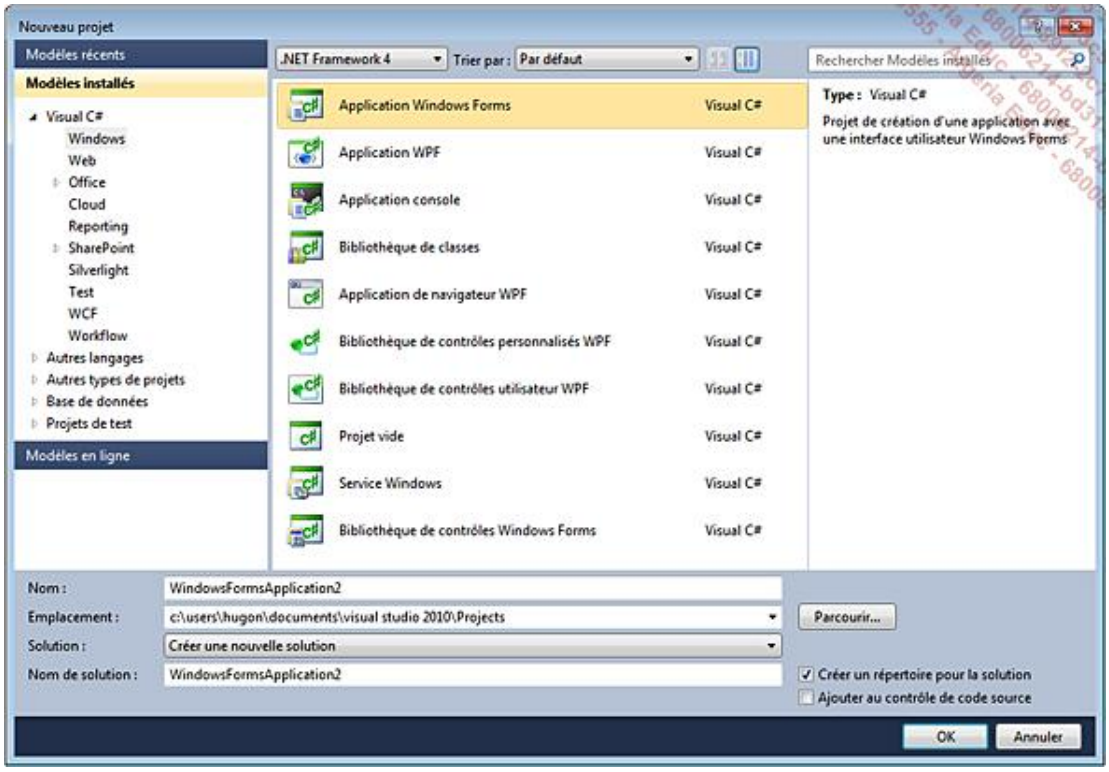
L'onglet **Extensions installées** permettra de désinstaller ou désactiver les extensions téléchargées.

La création de solutions

En travaillant avec Visual Studio, il est rare de commencer à partir d'une solution vide. Visual Studio propose des modèles de projets contenant les éléments par défaut, les références et configurations pour le type voulu.

La sélection du type de projet se fait lors de sa création. La boîte de dialogue suivante permet de faire la sélection :

- **Fichier - Nouveau - Projet ...**



Pour l'application qui sera développée au cours de cet ouvrage, nous allons créer un type de projet **Application Windows Forms** nommé **SelfMailer**.

Dès la validation du choix, Visual Studio ouvre la solution avec les fichiers de base, un formulaire **Form1.cs**, et un fichier qui est le point d'entrée principal de l'application **Program.cs**.

Pour le moment ce projet n'a aucune fonctionnalité. En l'exécutant (**F5**), le formulaire **Form1** vide apparaît. Visual Studio a déjà fait l'intégration de base lors de la création du projet en insérant le code suivant dans le fichier **Program.cs** :

```
namespace SelfMailer
{
    static class Program
    {
        /// <summary>
        /// Point d'entrée principal de l'application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false;
            Application.Run(new Form1());
        }
    }
}
```

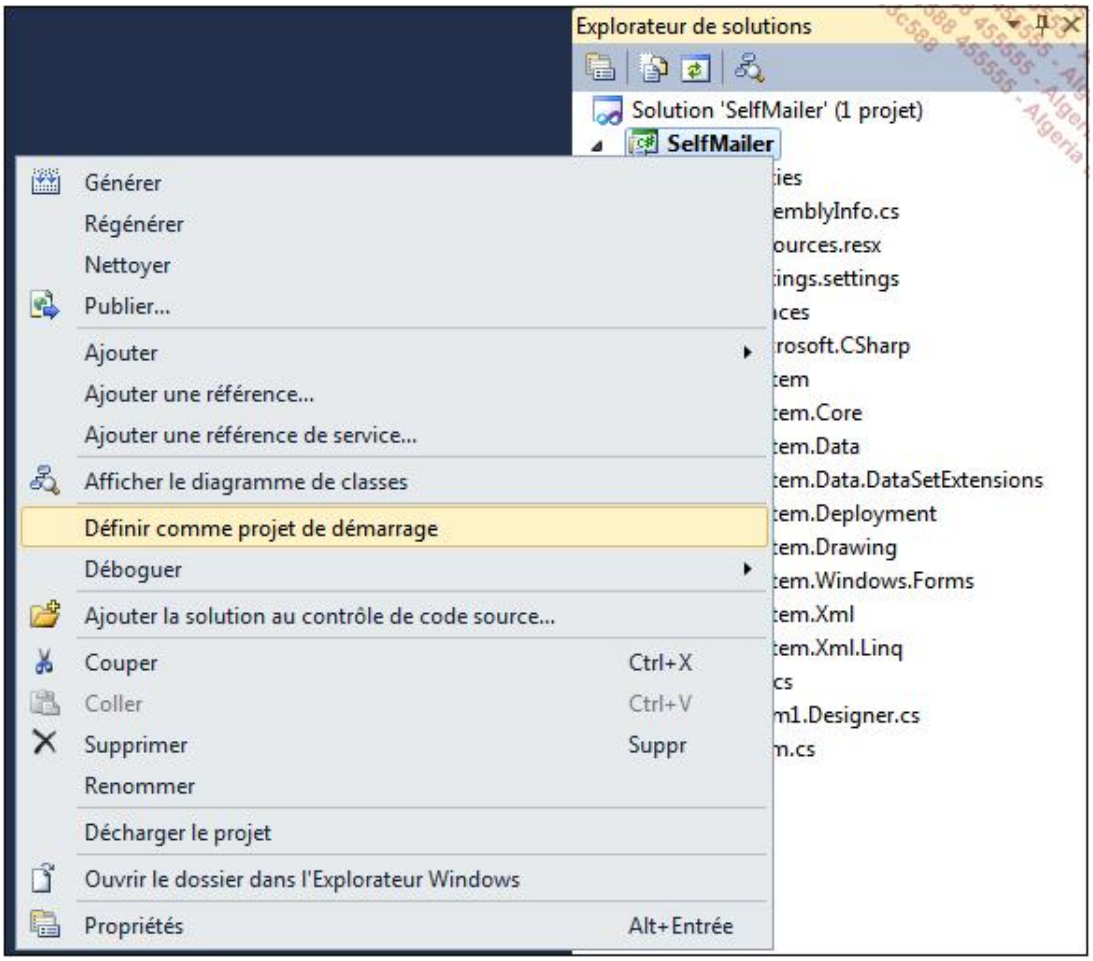
1. Définir le point d'entrée

Chaque application exécutable, par opposition aux bibliothèques de classes, doit posséder un point d'entrée. Ce point d'entrée est la méthode `Main`, ci-dessus, qui permet de générer le formulaire. Le point d'entrée est paramétrable via les propriétés du projet en définissant la propriété **Objet de démarrage**.

Cette méthode doit être publique et statique via les mots clés `public` et `static`, qui spécifient respectivement au compilateur que la méthode est accessible depuis l'application et en dehors, que la méthode est globale et que la classe n'a pas besoin d'être instanciée pour pouvoir l'appeler. À noter que si la méthode ne possède pas l'identificateur d'accès, elle l'héritera de sa classe qui est par défaut publique.

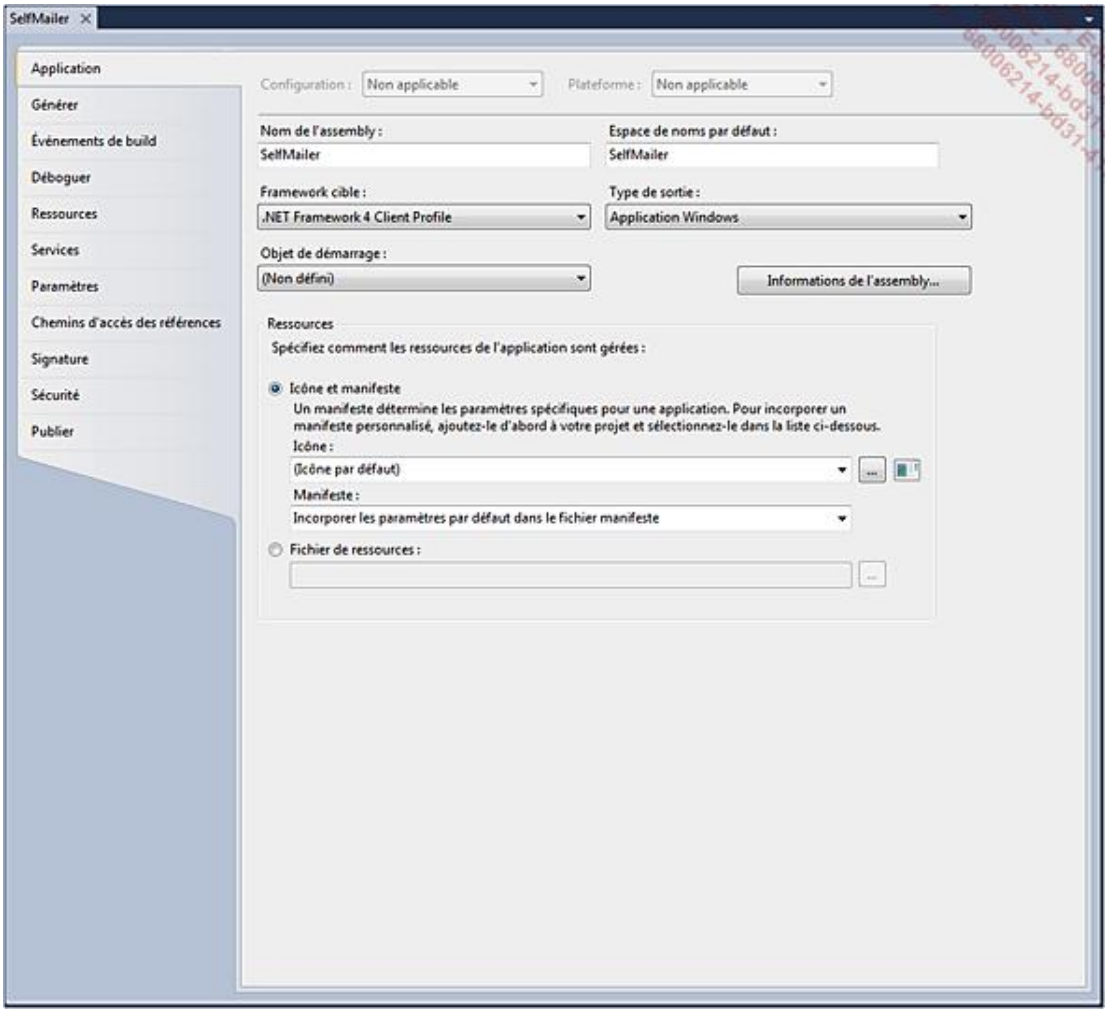
2. La différence entre projets et solutions

Un projet est un ensemble de fichiers qui seront compilés en un seul assemblage. Une solution est un ensemble d'un ou plusieurs projets. De la même manière qu'un projet à un point d'entrée, une solution a un projet de démarrage. Ce projet est identifiable dans l'explorateur de solutions car son nom est en gras. Pour modifier cette propriété, un clic avec le bouton droit sur le projet permet de sélectionner **Définir comme projet de démarrage** dans le menu contextuel :



3. Configurer le projet

La configuration du projet se fait en faisant un clic avec le bouton droit sur le projet dans l'explorateur de solutions, puis en sélectionnant le menu **Propriétés** du menu contextuel :



Cette fenêtre contient des onglets permettant de configurer les différentes parties du projet.

L'onglet **Application** concerne les propriétés générales de l'application :

- **Nom de l'assembly** : vous pouvez spécifier le nom de l'assemblage qui sera généré et suivant le type de projet, l'extension **.exe** ou **.dll** sera ajoutée.

- **Espace de noms par défaut** : définit l'espace de noms qui sera utilisé lors de la création d'une nouvelle classe dans le projet :

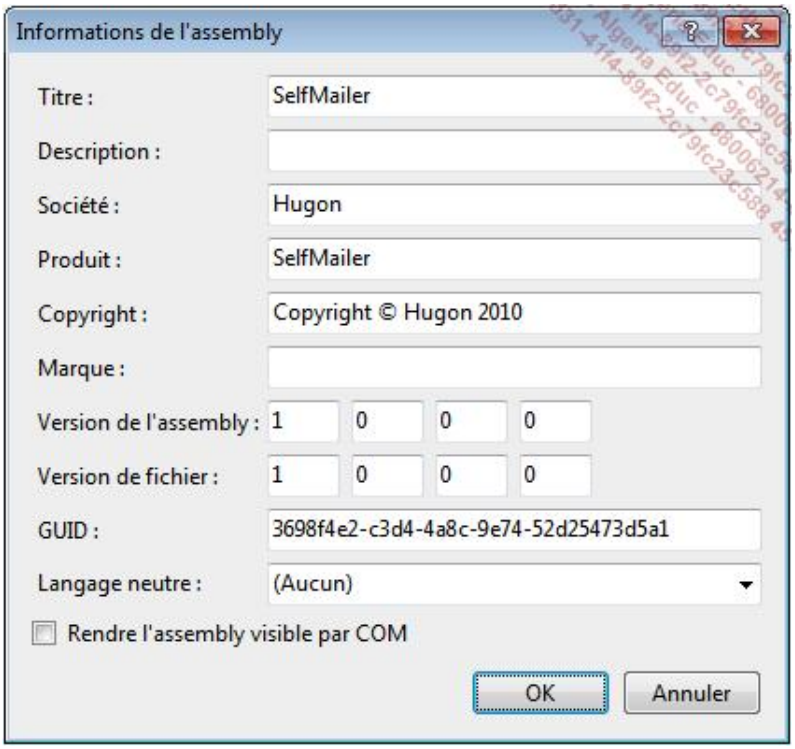
```
namespace SelfMailer
{
    ...
}
```

- **Framework cible** : permet de spécifier quelle version du Framework .NET sera utilisée pour générer la solution.

- **Type de sortie** : ce choix est fait en fonction du type d'application à produire, si cette propriété est modifiée après la création du projet, des changements devront être apportés au point d'entrée de l'application et à la méthode `Main()` ainsi qu'aux références du projet.

- **Objet de démarrage** : dans le cas où plusieurs méthodes `Main()` seraient présentes dans le projet, cette propriété permet de spécifier laquelle sera le point d'entrée.

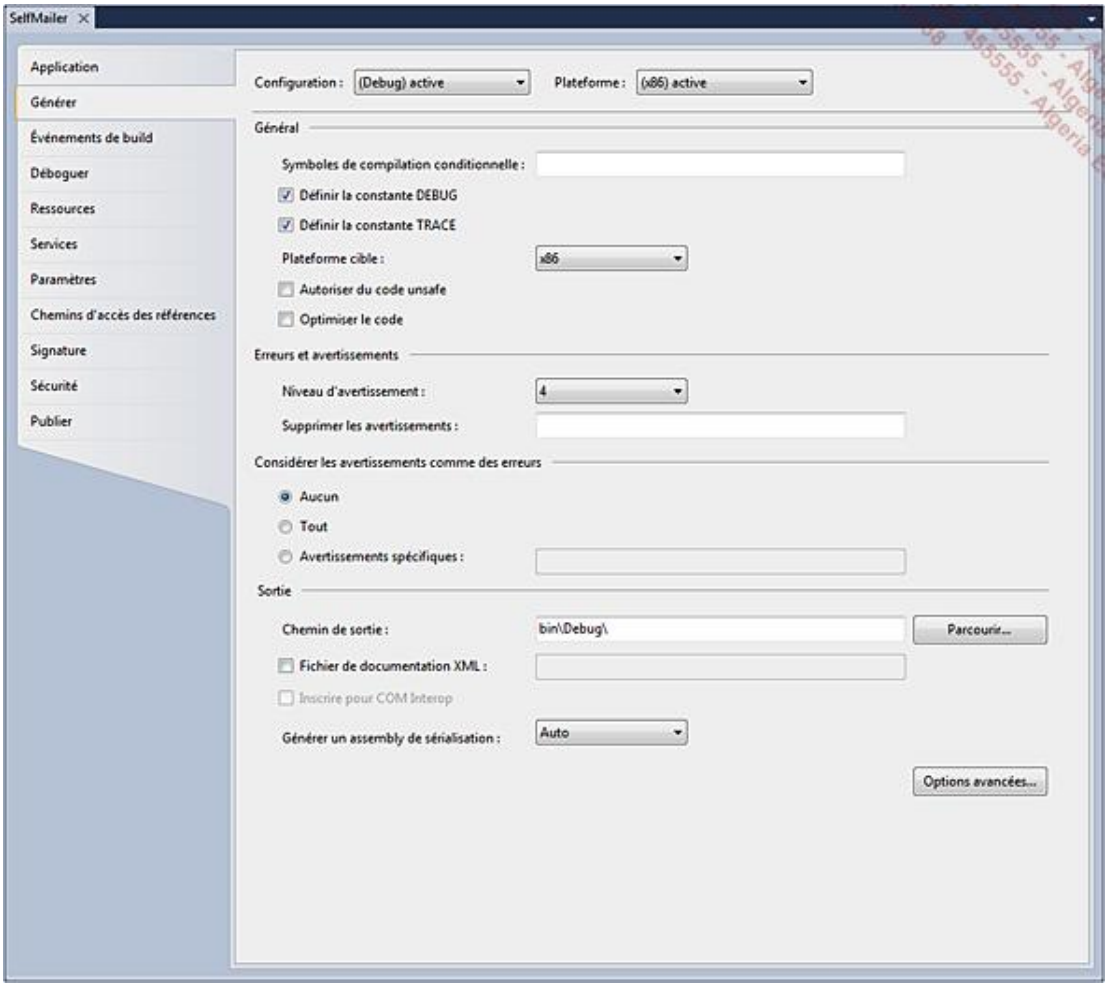
- **Informations de l'assembly** : ce bouton ouvre un formulaire permettant de spécifier des propriétés telles que le titre de l'application, le copyright, sa version :



Les informations de ce formulaire sont stockées dans le fichier **AssemblyInfo.cs** sous le dossier **Properties** du projet :

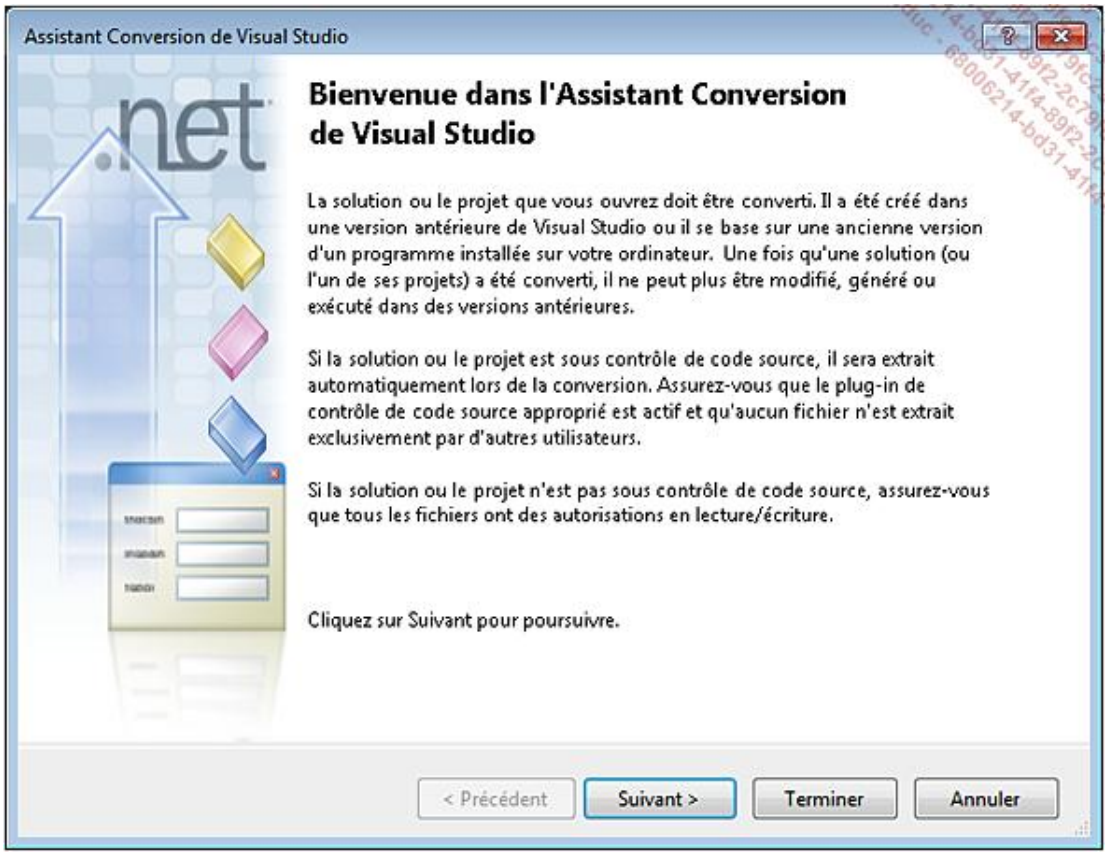
```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("SelfMailer")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Hugon")]
[assembly: AssemblyProduct("SelfMailer")]
[assembly: AssemblyCopyright("Copyright © Hugon 2010")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("3698f4e2-c3d4-4a8c-9e74-52d25473d5a1")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

L'onglet **Générer** spécifie les options de génération du projet. Par défaut le profil de génération est **Debug**. Utilisé en phase de conception, il permet de générer les symboles de débogage et ainsi de pouvoir utiliser le débuggeur intégré de Visual Studio. Lorsque le projet est terminé, la bonne pratique est de le compiler avec un profil **Release** qui ne contiendra pas ces symboles et sera donc plus léger mais ne pourra pas être utilisé par le débuggeur de Visual Studio :



4. La conversion de solutions

Les solutions créées avec les versions précédentes de Visual Studio peuvent être ouvertes avec la version 2010. Visual Studio lancera un avertissement indiquant que la solution sera mise à jour avec l'assistant de conversion :



Avant le lancement de la mise à jour de la solution, l'assistant propose d'effectuer une sauvegarde de la solution.

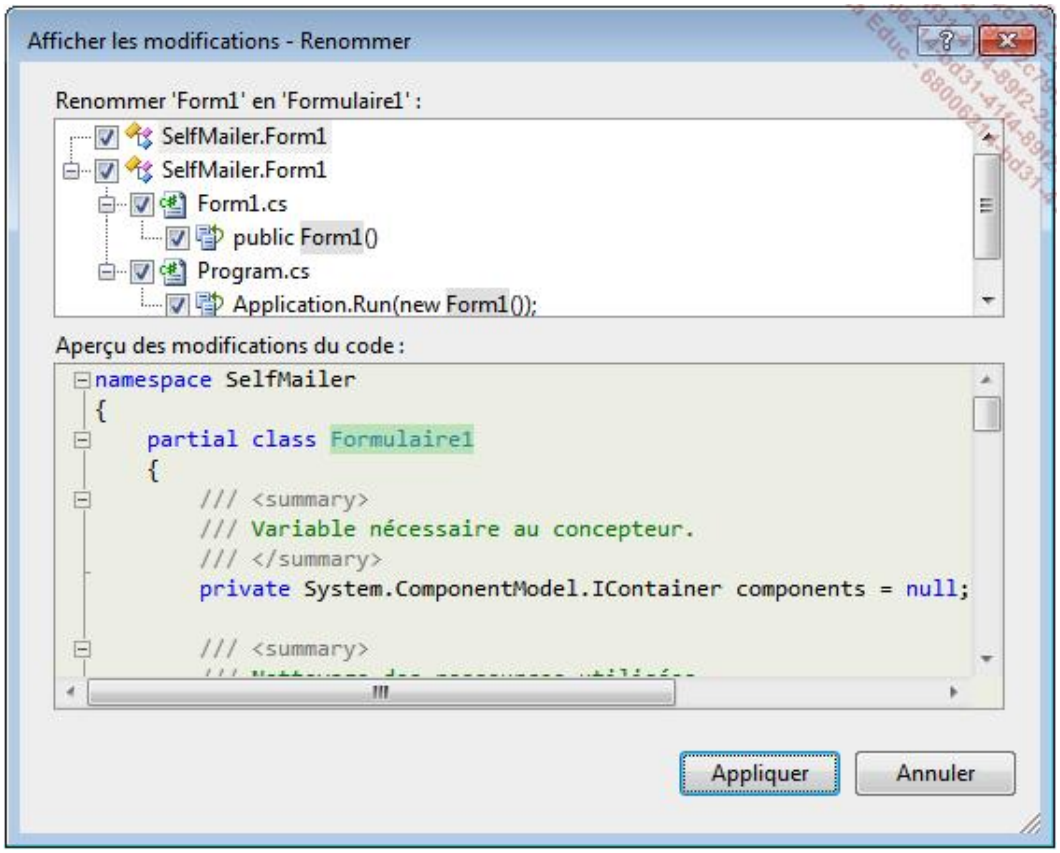
Lorsque la conversion est terminée, l'utilisateur a la possibilité d'afficher le journal de conversion afin de consulter les erreurs ou avertissements qui auraient pu être générés par la conversion.

5. Les outils de refactorisation

La refactorisation permet de modifier la structure du code sans en modifier le comportement. Les options de refactorisations sont disponibles via le menu contextuel dans l'éditeur de texte ou à partir du menu **Refactoriser** :

Plusieurs méthodes pour refactoriser le code sont disponibles :

- **Renommer** (**F2**): permet de renommer les variables, méthodes, classes ou espaces de noms. Visual Studio se charge de trouver les références dans le projet et de les remplacer. Avant d'appliquer les modifications, Visual Studio propose de vérifier les changements qui seront apportés :



- **Extraire la méthode** (**Ctrl+R, M**) : permet de créer une nouvelle méthode à partir d'une sélection de code à l'intérieur d'un bloc. La nouvelle méthode contient le code sélectionné qui est remplacé par un appel à cette nouvelle méthode.

- **Encapsuler le champ** (**Ctrl+R, E**) : permet de créer une propriété à partir d'une variable.

- **Extraire l'interface** (**Ctrl+R, I**) : génère un nouveau fichier d'interface regroupant les membres communs utilisés par plusieurs classes, structures ou interfaces.

- **Supprimer les paramètres** (**Ctrl+R, V**) : affiche une boîte de dialogue permettant de supprimer les paramètres d'une méthode. Si le paramètre est utilisé dans la méthode, il ne sera pas supprimé et Visual Studio signalera une erreur. Par contre, les paramètres supprimés le seront des appels de la méthode.

- **Réorganiser les paramètres** (**Ctrl+R, O**) : affiche une boîte de dialogue permettant de modifier l'ordre des paramètres d'une méthode. Les appels à la méthode seront modifiés pour que les paramètres tiennent compte du nouvel ordre :

```
public int add(int i, int j)
{
    return i + j;
}
// Appel de la méthode:
add(1, 2);
```

Après l'inversion des paramètres i et j dans la signature de la méthode, l'appel sera modifié par Visual Studio de la manière suivante :

```
add(2, 1);
```

Introduction

Le Framework .NET est l'élément central des applications. Il gère l'exécution, l'allocation mémoire, les permissions. L'architecture .NET est principalement constituée de deux composants : d'une part le CLR (*Common Language Runtime*) et d'autre part les bibliothèques de classes.

CLR

Le CLR est l'environnement d'exécution des applications. Il est multilangage grâce au CLS (*Common Language Specification*) qui est un ensemble de règles à respecter pour un compilateur créant des applications à exécuter avec le CLR. La grande force du CLR est de pouvoir combiner plusieurs assemblages quel que soit le langage dans lequel ils ont été écrits. Une application écrite en C# pourra ainsi faire référence et utiliser une librairie écrite en VB.

Pour en arriver à ce niveau de compatibilité, le compilateur convertit le code en langage intermédiaire (IL) permettant d'être interprété de la même manière quel que soit le langage dans lequel le code a été écrit. La compatibilité des types entre les différents langages est assurée par le CTS (*Common Types System*). Chaque type de base d'un langage possède un équivalent dans le Framework .NET et donc en langage intermédiaire. Ainsi, un `integer` en VB et un `int` en C# seront du même type `System.Int32`.

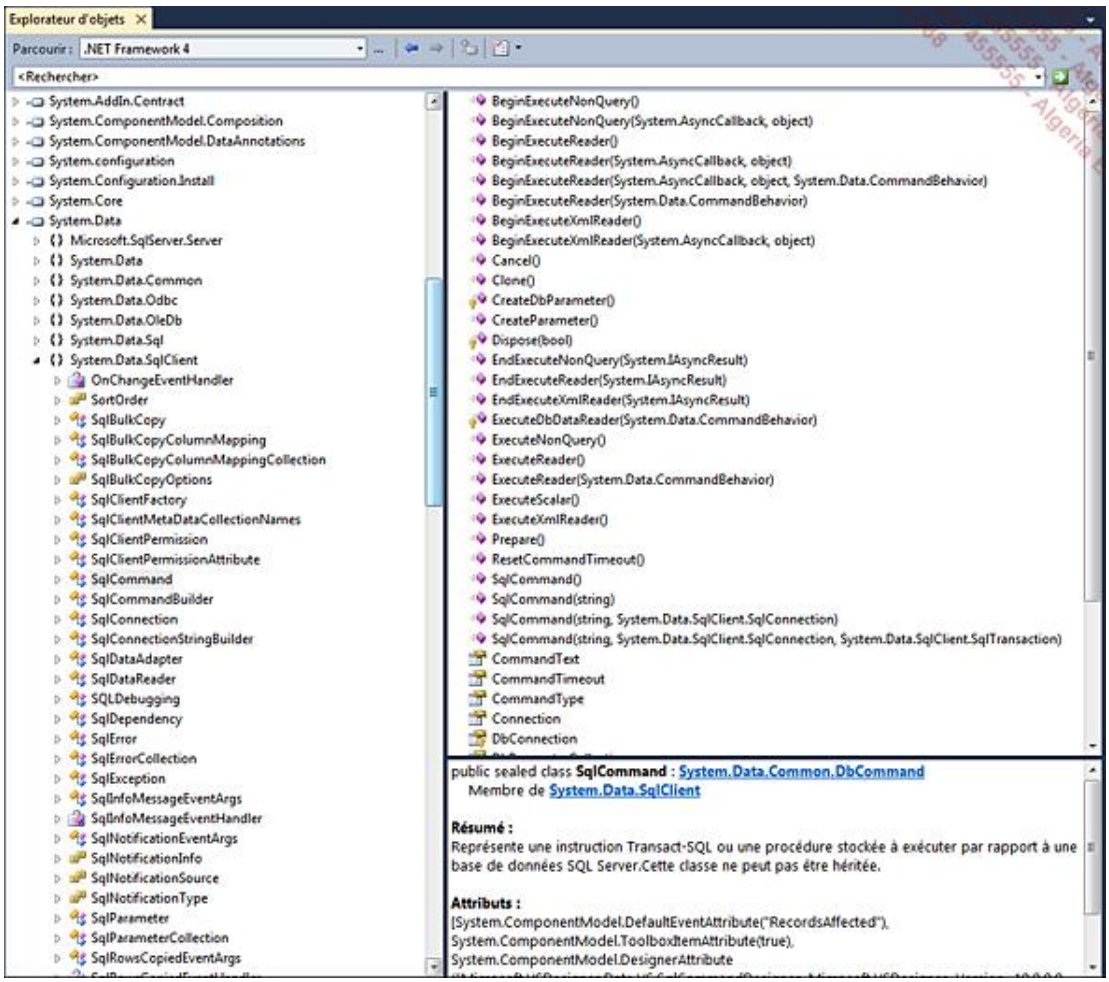
Une fois compilée, une application se résume à au moins un fichier exécutable. Celui-ci est en langage intermédiaire. Lorsque l'exécution est lancée, le CLR examine le manifeste pour déterminer si les conditions de sécurité sont respectées. Si tel est le cas, le CLR crée un processus pour héberger l'application et le code est compilé du langage intermédiaire en code binaire par le compilateur JIT (*Just In Time*). Ce code généré est ensuite stocké en mémoire de manière à ne pas être recompilé en cours d'exécution et à optimiser les performances.

Les bibliothèques de classes

Le Framework .NET est composé de plusieurs bibliothèques de classes, classées en espaces de noms, organisés de manière hiérarchique à partir de l'espace de noms racine `System`. Les fonctionnalités en relation sont donc classées au sein d'un même espace de noms. `System.IO`, par exemple, regroupe les types ayant pour but d'interagir avec le système de fichiers. Voici quelques espaces de noms couramment utilisés :

Espace de noms	Description
`System`	Espace de noms racine. Il contient les types de base du Framework .NET.
`System.Collections`	Contient les types permettant de créer et gérer les listes et les tableaux.
`System.Data`	Contient les types requis pour la manipulation et la communication avec les bases de données.
`System.Drawing`	Contient l'ensemble des types permettant de gérer le rendu graphique et le traitement d'éléments visuels.
`System.IO`	Contient l'ensemble des classes permettant d'interagir avec le système de fichiers.
`System.Math`	Fournit les types permettant d'exécuter des calculs plus complexes que de simples opérations.
`System.Windows.Forms`	Contient tous les types permettant la création d'applications Windows, que ce soit les formulaires ou les contrôles.

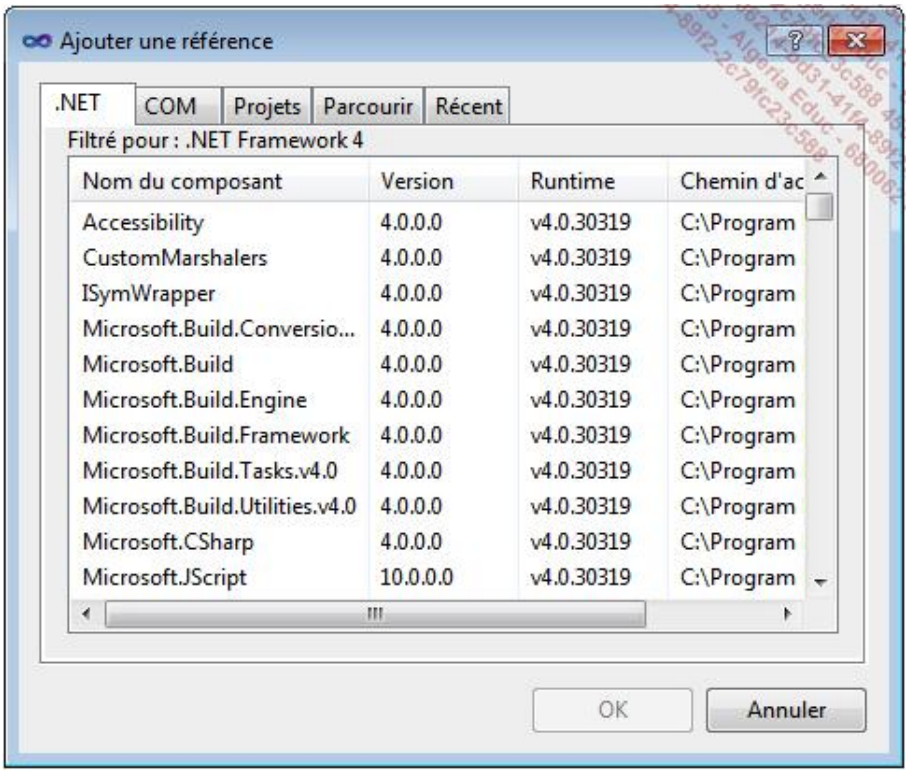
Ces bibliothèques de classes sont consultables via l'explorateur d'objets :



- **Affichage - Explorateur d'objets**

Il n'est pas rare d'avoir besoin de bibliothèques de classes autres que celles fournies par le Framework .NET, pour pouvoir les utiliser dans un projet. Il suffit d'ajouter une référence vers la librairie de classes.

Dans l'explorateur de solutions, un clic avec le bouton droit sur le nœud **Références** du projet permet d'ouvrir le menu contextuel. Le menu **Ajouter une référence...** ouvre la fenêtre de sélection :



Il est possible de choisir une référence vers les classes .NET, COM, à partir d'un autre projet de la solution ou de toute autre librairie de classes.

Les types

Le Framework .NET est composé de deux sortes de types : les types valeur et les types référence. Pour stocker les données, le CLR dispose de deux zones de mémoire : la pile et le tas.

La pile a un fonctionnement du type : dernier entré, premier sorti. Cette zone va héberger en mémoire les variables du programme, ainsi lorsqu'une fonction est appelée, ses paramètres sont stockés dans la pile et si cette fonction en appel d'autres, les variables sont placées au dessus des précédentes. À la fin de l'exécution de la fonction, les variables deviennent hors de portée et sont retirées de la pile libérant la mémoire occupée.

Le tas est une zone réservée au stockage des objets réutilisables. Le CLR gère les entrées et sorties de cet espace mémoire de manière automatique grâce à la récupération de la mémoire. Ce processus étant toujours actif, il est inutile de détruire explicitement les objets au sein de l'application. Le récupérateur s'exécute dans un processus de faible priorité qui peut augmenter si la mémoire a besoin d'être vidée Splus rapidement. L'inconvénient principal est qu'il n'est pas possible de connaître exactement le moment où un objet sera détruit et par conséquent, le moment d'exécution du code associé à cette destruction n'est pas maîtrisable. Les classes qui ont besoins de beaucoup de ressources peuvent implémenter une méthode `Dispose()` afin de les libérer.

1. Les types valeur

L'espace mémoire utilisé par une donnée de type valeur est affecté à la pile pendant sa durée de vie, puis, lorsque la variable est hors de portée, l'espace mémoire est libéré.

Les types valeur sont classiquement les types de base tels que les booléens (`bool`), les entiers (`int`) ou encore les caractères (`char`) parmi d'autres. Les structures (`struct`) et les énumérations (`enum`) en font également partie.

Un type valeur contient réellement la donnée qu'il représente en mémoire, il ne peut donc, par conséquent, pas être vide (`null`).

L'instruction suivante déclare une variable de type entier (`int`) et lui affecte une valeur. Le compilateur lève une exception si un type valeur est utilisé sans avoir été affecté :

```
int i = 1;
```

La déclaration de cette variable entraîne la réservation d'un espace mémoire de 32 bits et lui affecte la valeur 1.

Examinons l'exemple suivant :

```
bool b1 = true;
bool b2 = b1;
b1 = false;
```

Une première variable (b1) de type booléen est créée avec la valeur `true`. Une seconde variable (b2) de type booléen est créée avec la valeur de b1. Pour finir la variable b1 est modifiée. Quelle est la valeur de la variable b2 ? La réponse n'est pas surprenante : la variable b2 a la valeur `true`. b1 et b2 sont deux variables distinctes et la modification de l'une n'affecte pas la seconde. La seconde ligne qui affecte la valeur de b1 à la variable b2 entraîne, au niveau mémoire, la copie de la valeur sans autre lien.

2. Les types référence

Les types référence ont un fonctionnement différent au niveau de la mémoire. Les données réelles du type sont stockées dans le tas et un pointeur vers ces données est stocké dans la pile. Ainsi lors de l'utilisation d'une variable de type référence, ce n'est pas la valeur qui est passée mais le pointeur de celle-ci. Lorsque la variable est hors de portée, comme pour un type valeur, l'espace mémoire de la pile est libéré mais pas l'objet lui-même qui se trouve dans le tas. Lorsque la pile ne contient plus aucun pointeur vers cet objet, il devient récupérable par le mécanisme de récupération automatique de la mémoire.

Les classes, les interfaces et les délégués représentent des types référence et la manipulation des objets créés à partir de ces types se fait indirectement via une référence vers leur espace mémoire alloué.

Un type référence, à la différence d'un type valeur, doit être instancié grâce au mot clé `new`. Ce mot clé fait appel au constructeur du type. Il s'agit d'une méthode particulière qui initialise l'objet par l'affectation des valeurs aux membres. Pour certaines classes, il est possible de passer des paramètres au constructeur suivant les surcharges disponibles. Ainsi, il est possible de déclarer un `ArrayList` (de l'espace de noms `System.Collections`) de ces deux manières :

```
ArrayList tab1 = new ArrayList();
ArrayList tab2 = new ArrayList(1);
```

La première instanciation crée un objet de type `ArrayList` avec les valeurs par défaut. La seconde instanciation crée un même objet avec une capacité maximale de 1.

Un type référence peut avoir une valeur vide (`null`) et donc être déclaré mais non instancié.

Pour illustrer le fonctionnement en mémoire d'un type référence, prenons la classe suivante :

```
class Class
{
    public bool b;
}
```

Réalisons les opérations suivantes avec cette classe :

```
Class C1 = new Class();
C1.b = true;
Class C2 = C1;
C1.b = false;
```

Un premier objet est instancié. La valeur `true` est ensuite affectée à sa propriété b. L'objet C1 est ensuite affecté à un second objet du même type, puis la propriété b de l'objet C1 est modifiée. Quelle est la valeur de la propriété b de l'objet C2 ? La réponse n'est pas aussi évidente que pour un type valeur. En effet lors de l'affectation de l'objet C1 à l'objet C2, il y a eu une copie de la référence qui se trouve dans la pile du premier objet vers le second. Cela signifie que les deux objets pointent vers la même adresse mémoire, dans le tas. Donc la modification d'une valeur dans l'un des objets affecte les deux.

La syntaxe

1. Les identifiants

Les identifiants sont les noms donnés aux classes et à leurs membres. Un identifiant doit être composé d'un seul mot commençant par une lettre ou un caractère underscore (_). Les identifiants peuvent être composés de lettres majuscules ou minuscules mais le langage C# étant sensible à la casse, les majuscules et minuscules doivent être respectées pour faire référence au bon identifiant.

`monIdentifiant` est différent de `MonIdentifiant`.

2. Les mots clés

Les mots clés sont des noms réservés par le langage C#. Ils sont interprétés par le compilateur et ne peuvent donc pas être utilisés en tant qu'identifiants. Ces mots clés sont distingués dans l'éditeur de texte de Visual Studio en étant colorés en bleu (avec les paramètres d'apparence par défaut).

Si vous avez besoin d'utiliser un mot clé en tant qu'identifiant pour un membre, il faut préfixer le nom de l'identifiant par le caractère `@`. La syntaxe suivante entraînera une erreur et le compilateur refusera de s'exécuter :

```
private bool lock;
```

En préfixant le membre `lock` par le caractère `@`, le compilateur considère que c'est un identifiant et non plus un mot clé :

```
private bool @lock;
```

Le caractère `@` peut également préfixer des identifiants qui n'ont aucun conflit avec les mots clés, ainsi `@monIdentifiant` sera interprété de la même manière que `monIdentifiant`.

Voici une liste des mots clés du langage C#. Ils seront expliqués, en partie, au cours de l'ouvrage :

abstract	add	as	ascending	base
bool	break	by	byte	case
catch	char	checked	class	const
continue	decimal	default	delegate	descending
do	double	dynamic	else	equals
enum	event	explicit	extern	false
finally	fixed	float	for	foreach
from	get	global	goto	group
if	implicit	in	int	interface
internal	into	is	join	let
lock	long	namespace	new	null
object	on	operator	orderby	out
override	params	partial	private	protected
public	readonly	ref	remove	return

sbyte	sealed	select	set	short
sizeof	stackalloc	static	string	struct
switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe
ushort	using	value	var	virtual
volatile	void	where	while	yield

3. La ponctuation

La ponctuation a pour objectif de séparer les instructions du programme de manière logique, compréhensible par l'humain et interprétable par le compilateur.

Toute instruction doit se terminer par un point-virgule (;). S'il est oublié à la fin de l'instruction, le compilateur lève une erreur de syntaxe. L'avantage, cependant, est de pouvoir écrire une instruction sur plusieurs lignes :

```
int i = 5
        + 2;
```

Le point (.) après un identifiant permet d'accéder aux membres d'un objet. Visual Studio affiche, grâce à l'IntelliSense, la liste des membres disponibles dès que le point est ajouté à un objet :

```
monObjet.maPropriete
```

Les accolades ({ et }) sont utilisées pour grouper plusieurs instructions au sein d'un bloc de contrôle ou d'une méthode. Elles indiquent à quel endroit les instructions commencent et à quel endroit elles se terminent :

```
class Program
{
}
```

Les parenthèses ((et)) sont utilisées pour la déclaration ou l'appel de méthodes. Elles peuvent contenir des paramètres suivant la signature de la méthode. Les paramètres d'une méthode sont séparés par une virgule (,) :

```
monObjet.maMethode(Parametre1, Parametre2);
```

Les parenthèses sont également utilisées pour grouper les instructions de la même manière que pour une opération mathématique.

Les crochets ([et]) permettent d'accéder à l'élément d'un tableau ou, si la classe contient une propriété indexeur, à l'élément d'une classe. Par exemple, si la classe `monObjet` est un tableau de valeurs de type `string`, pour accéder à son premier élément, la syntaxe serait la suivante :

```
string s = monObjet[0];
```

Les éléments des tableaux sont indexés à partir de 0.

4. Les opérateurs

a. Les opérateurs de calcul

Les opérateurs de calcul permettent, comme en mathématique, d'effectuer des opérations.

L'addition est réalisée avec l'opérateur + :

```
i = 5 + 2;        // i = 7
```

La soustraction est réalisée avec l'opérateur - :

```
i = 5 - 2;        // i = 3
```

La multiplication est réalisée avec l'opérateur * :

```
i = 5 * 2;        // i = 10
```

La division est réalisée avec l'opérateur / :

```
i = 6 / 2;        // i = 3
```

Le modulo est réalisé avec l'opérateur % :

```
i = 5 % 2;        // i = 1
```

b. Les opérateurs d'assignation

Les opérateurs d'assignation permettent d'assigner une valeur à une variable. L'opérateur le plus utilisé est le caractère = :

```
i = x;
```

Il est également possible de réaliser une assignation et un calcul en même temps en combinant deux opérateurs :

```
i += 1;
```

en utilisant l'opérateur +=, il y a affectation à la variable de sa propre valeur additionnée de la valeur à droite de l'opérateur. Cette instruction est équivalente à celle-ci :

```
i = i + 1;
```

La combinaison d'opérateurs de calcul et d'assignation est valable pour tous les opérateurs :

```
int i = 5;
i += 2;          // i = 7
i -= 2;          // i = 5
i *= 2;          // i = 10
i /= 2;          // i = 5
i %= 2;          // i = 1
```

c. Les opérateurs de comparaison

Les opérateurs de comparaison sont essentiellement utilisés dans le cadre de décisions au sein d'instructions de contrôles.

L'opérateur == détermine si deux variables sont égales :

```
x == y           // renvoie true si x égal y
```

L'opérateur != détermine si deux variables sont différentes :

```
x != y           // renvoie true si x est différent de y
```

L'opérateur > détermine si la variable de gauche est strictement supérieure à la variable de droite :

```
x > y            // renvoie true si x est supérieur à y
```

L'opérateur >= détermine si la variable de gauche est supérieure ou égale à la variable de droite :

```
x >= y           // renvoie true si x est supérieur ou égal à y
```

L'opérateur < détermine si la variable de gauche est strictement inférieure à la variable de droite :

```
x < y            // renvoie true si x est inférieur à y
```

L'opérateur <= détermine si la variable de gauche est inférieure ou égale à la variable de droite :

```
x <= y          // renvoie true si x est inférieur ou égal à y
```

L'opérateur (et mot clé) `is` permet de déterminer le type d'un objet :

```
x is int        // renvoie true si x est du type int
```

Il est également possible de combiner les opérateurs de comparaison avec des opérateurs logiques. L'opérateur `&&` permet de spécifier un ET logique tandis que l'opérateur `||` spécifie un OU logique. Les différentes expressions peuvent être combinées grâce à des parenthèses afin de modifier l'ordre d'interprétation :

```
(x >= 0 || x > 10) && (x <= 1 || x < 25)
```

5. La déclaration de variables

La déclaration de variables se fait en spécifiant sont type puis en indiquant son identifiant. L'instruction suivante déclare une variable nommée `s` et du type `string` :

```
string s;
```

L'instruction de déclaration se termine comme toutes les instructions par le point virgule.

Une variable peut être déclarée et initialisée avec la même instruction :

```
string s = "La valeur de ma variable";
```

Il est également possible de déclarer et d'initialiser plusieurs variables en une seule instruction, à la condition qu'elles soient de type identique. Les variables sont séparées par une virgule :

```
bool b1 = true, b2 = false;
```

Une variable peut également être marquée avec le mot clé `const` qui spécifie que la valeur de la variable ne peut pas être modifiée pendant l'exécution. C'est une variable en lecture seule :

```
const int i = 0;
```

La portée d'une variable déclarée dans le corps d'une méthode se limite à celle-ci. C'est-à-dire qu'elle est détruite dès la fin de l'exécution de la méthode. Elle sort de la portée.

Les variables déclarées au sein d'un bloc conditionnel ou itératif ne sont accessibles que dans ce bloc. Pour les blocs itératifs, à la fin de la boucle la variable est détruite puis elle est à nouveau initialisée au passage suivant.

6. Les instructions de contrôles

a. Les instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter une portion de code en fonction de tests effectués sur les variables de l'application. Il existe deux types de structures conditionnelles : les blocs *if* et les blocs *switch*.

if, else et else if

Syntaxe générale :

```
if (expression)
{
       instructions
}
[else if (expression)
{
       instructions
}]
```

```
[else
{
        instructions
}]
```

L'instruction `if` évalue une expression booléenne et exécute le code si cette expression est vraie (`true`). Sa syntaxe est la suivante :

```
if (x > 10)
{
   // Instructions exécutées si x est supérieur à 10
}
```

L'expression à évaluer est insérée entre parenthèses après le mot clé `if` et doit avoir un résultat de type booléen.

L'instruction `else` permet d'intégrer le code qui sera exécuté si l'expression évaluée dans l'instruction if est fausse (`false`) :

```
if (x > 10)
{
   // Instructions exécutées si x est supérieur à 10
}
else
{
   // Instructions exécutées si x est inférieur ou égal à 10
}
```

L'instruction `else if` permet d'évaluer une nouvelle expression lorsque celle de l'instruction `if` est fausse. Il peut y avoir plusieurs instructions `else if` mais un bloc de décision `if` doit toujours commencer par une instruction `if` facultativement suivi de une ou plusieurs instructions `else if` et terminer de manière non obligatoire par une instruction `else` :

```
if (x > 10)
{
   // Instructions exécutées si x est supérieur à 10
}
else if (x = 10)
{
   // Instructions exécutées si x est égal à 10
}
else
{
   // Instructions exécutées si x est inférieur à 10
}
```

Les instructions au sein de la structure de décision sont entourées par des accolades indiquant le début et la fin du bloc. Au cas où le bloc ne contiendrait qu'une seule ligne, il est permis de les omettre (valable aussi pour les autres instructions de contrôles) :

```
if (x > 10)
   x -= 10;
else if (x = 10)
   x -= 1;
else
   x += 10;
```

switch

Syntaxe générale :

```
switch (expression)
{
        case expression ou constante:
                instructions
                [instructions de saut]
        [default:
                instructions]
}
```

Une instruction `switch` peut être considérée comme une suite d'instructions `if`, `else if` et `else`.

L'instruction `switch` évalue la valeur d'une variable passée en paramètre et exécute les instructions en fonction des valeurs possibles :

```
switch (i)
{
    case 0:
        i++;
        break;
    case 1:
        i--;
        break;
    default:
        i = 0;
        break;
}
```

Le mot clé `default` doit être placé en dernier dans la chaîne d'évaluation et le code sera exécuté si aucune des conditions précédentes n'est remplie. Ajouter l'instruction `default` est facultatif.

Chaque bloc d'instructions au sein d'une évaluation doit contenir le mot clé `break` afin de sortir du bloc `switch`. S'il n'est pas présent, les autres conditions seront évaluées et potentiellement exécutées. Les autres instructions de saut sont également valables au sein d'une structure `switch`.

Il est possible de combiner plusieurs évaluations si celles-ci doivent exécuter les mêmes instructions :

```
switch (i)
{
    case 0:
    case 1:
    case 2:
        i++;
        break;
    case 3:
        i--;
        break;
}
```

b. Les instructions itératives

Les instructions itératives permettent d'effectuer des boucles sur une série d'instructions en fonction de l'évaluation d'une expression. Les instructions itératives sont les boucles `for`, `while`, `do while` et `foreach`.

for

Syntaxe générale :

```
for (initialisation; expression; pas)
{
        instructions
}
```

Une boucle `for` contient une série d'instructions qui seront exécutées plusieurs fois suivant les paramètres d'initialisation, d'expression et de pas. L'initialisation est la déclaration et l'affectation d'une variable de type numérique. À chaque passage dans la boucle, la variable est incrémentée de la valeur du pas et l'expression est évaluée afin de déterminer si les instructions doivent être exécutées ou non.

```
int total = 0;
for (int i = 0; i < 4; i++)
{
    total += i;
}
// total = 6
```

Le code précédent déclare une variable `total` et lui assigne la valeur 0. Une boucle `for` déclare une variable `i` en lui

affectant la valeur 0. L'expression est ensuite évaluée et si elle est vraie, les instructions sont exécutées. Lorsque les instructions sont toutes exécutées, la variable i est affectée par le pas. Ici il s'agit d'ajouter 1 à la variable `i`. L'expression est à nouveau évaluée et la boucle continue jusqu'à ce que l'expression soit fausse.

La variable déclarée dans l'instruction `for` peut être modifiée dans le corps de la boucle :

```
for (int i = 0; i < 4; i++)
{
    i++;
}
// Le corps de la boucle for est exécuté deux fois
```

Il est également possible de définir un pas négatif :

```
for (int i = 10; i > 0; i--)
{}
```

while

Syntaxe générale :

```
while (expression)
{
      instructions
}
```

Une boucle `while` permet d'exécuter une série d'instructions tant que l'expression évaluée est vraie :

```
int i = 0;
while (i < 4)
{
    i++;
}
// i = 3
```

do while

Syntaxe générale :

```
do
{
       instructions
}
while (expression);
```

La boucle `do while` est similaire à la boucle `while` dans le sens où les instructions de la boucle sont exécutées si l'évaluation de l'expression est vraie. La différence réside dans le fait que la boucle `do while` effectue la vérification de l'expression après l'exécution des instructions. La boucle `do while` garantit que les instructions seront exécutées au moins une fois :

```
int i = 0;
do
{
    i--;
}
while (i >= 0);
// i = -1
```

foreach

Syntaxe générale :

```
foreach (type nom in object)
{
       instructions
```

```
}
```

Les boucles `foreach` sont utiles pour parcourir les tableaux, les collections et tous les types énumérables. L'instruction déclare un type et parcourt l'objet passé en paramètre pour le type donné. La boucle s'exécute tant qu'il y a des éléments dans l'objet :

```
string s1 = "foreach";
string s2 = "";
foreach (char c in s1)
{
    s2 += c;
}
// s2 = "foreach"
```

La variable `s1` de type `string` est composée d'une collection de type `char`. La boucle `foreach` les prend un par un et les affecte à la variable `c` de type `char`. La variable `c` peut ensuite être utilisée dans le corps de la boucle.

c. Les instructions de saut

Les instructions de saut permettent de modifier le flux d'exécution du programme. Les instructions de saut sont `break`, `continue`, `goto`, `return` et `throw`.

break

L'instruction `break` permet de sortir d'une boucle ou d'une instruction `switch` sans attendre la fin de l'exécution de toutes les instructions :

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break;
    }
}
// i = 5, le corps de la boucle a été exécuté 5 fois
```

continue

L'instruction `continue` permet d'arrêter l'exécution du corps d'une boucle et de passer à la prochaine évaluation :

```
for (int i = 0; i < 10; i++)
{
    if (i > 5)
    {
        continue;
    }
}
// i = 5, le corps de la boucle a été exécuté 10 fois
```

goto

L'instruction `goto` permet de transférer l'exécution à un bloc d'instructions. Cette instruction est héritée des anciens langages procéduraux tels que le basic.

Une instruction `goto` est suivie par le nom d'une étiquette et l'exécution du code est transférée à celle-ci. L'étiquette est définie dans le code par un nom suivi de deux points (`:`) :

```
    int i = 0;
ajout:
    i++;

    if (i < 5)
    {
        goto ajout;
    }
// i = 5
```

L'instruction `goto` est également utilisée au sein des instructions `switch` pour passer d'un cas à un autre. `goto` peut référencer un autre cas ou le cas par défaut :

```
int i = 0;
switch (i)
{
    case 0:
        i++;
        goto 1;
    case 1:
        i--;
        goto default;
    default:
        i = 0;
        break;
}
```

return

L'instruction `return` est utilisée au sein d'une fonction et permet de définir la valeur de retour de celle-ci :

```
int Addition(int i, int j)
{
    return i + j;
}
```

throw

L'instruction `throw` permet de lever des exceptions. L'exécution est immédiatement interrompue et la nature de l'erreur est passée à l'appelant dans la pile d'appel jusqu'à être interceptée et traitée :

```
throw new Exception();
```

La gestion des exceptions sera étudiée plus en détail dans le chapitre consacré à la gestion des erreurs.

7. Les commentaires

Le code peut contenir des commentaires afin de faciliter sa compréhension s'il devait être étudié par un autre développeur ou si de la maintenance était requise.

Les commentaires commencent par le double signe `//`. Ils peuvent être placés sur une ligne seule ou à la fin d'une instruction, comme dans les précédents exemples. Ce type de commentaire est pour une ligne unique. Un retour à la ligne implique une nouvelle instruction.

Pour insérer des commentaires sur plusieurs lignes, ils doivent commencer par les caractères `/*` et se finir par les caractères `*/` :

```
/* Mon commentaire
sur plusieurs
lignes */
```

L'éditeur de texte de Visual Studio colorise les commentaires en vert (paramètre par défaut).

Bien plus que pour laisser des annotations, les commentaires peuvent également servir pour fournir une documentation au format XML. Cette documentation XML pourra être interprétée par l'IntelliSense pour fournir la description des membres, des paramètres ou encore des exceptions. Des outils tiers permettent également d'interpréter ces documentations XML afin de produire des fichiers d'aide sous différentes formes comme des pages HTML ou des fichiers CHM.

Un commentaire de documentation est placé directement avant le membre qu'il décrit et commence avec la suite de caractères `///`. En tapant cette suite de caractères dans Visual Studio, l'éditeur de texte implémente la base du commentaire. Saisissez la suite `///` avant le constructeur du formulaire Form. Visual Studio ajoute les éléments XML de base :

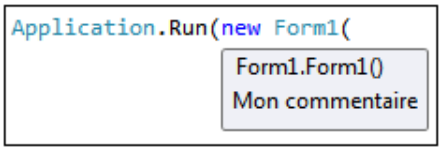
```
/// <summary>
///
```

```
/// </summary>
public Form1()
{
    InitializeComponent();
}
```

En plaçant un contenu entre les balises `summary` comme ceci :

```
/// <summary>
/// Mon commentaire

/// </summary>
```

Il sera interprété par l'IntelliSense de Visual Studio :



Voici une liste des balises les plus utilisées et reconnues pour la documentation XML :

Balise	Syntaxe	Description

`summary`	`<summary>`
	`...`
	`</summary>`
		Indique ce que l'info bulle de l'IntelliSense doit afficher.

`remarks`	`<remarks>`
	`...`
	`</remarks>`
		Texte supplémentaire de description du type ou du membre.

`param`	`<param name="`*`NomParamètre`*`">`
	`...`
	`</param>`
		Texte explicatif sur le paramètre attendu. Cette balise est utilisée dans la description de méthodes ou fonctions ayant des paramètres dans leur signature.

`returns`	`<returns>`
	`...`
	`</returns>`
		Texte explicatif sur la valeur de retour. Cette balise est utilisée dans le cadre des fonctions.

`exception`	`<exception [cref="type"]>`
	`...`
	`</exception>`
		Texte décrivant les exceptions et leur type qui peuvent être levées par la méthode ou fonction. L'attribut optionnel `cref` fait référence au type de l'exception.

`permission`	`<permission`
	`[cref="type"]>`
	`...`
	`</permission>`
		Texte décrivant les permissions nécessaires pour exécuter le code de la méthode. L'attribut optionnel `cref` fait référence au type de la permission.

`example`	`<example>` `...`	Texte présentant un exemple d'utilisation de la méthode ou fonction.

	</example>	

La génération du fichier de documentation XML au moment de la compilation est définie dans la fenêtre des propriétés du projet sous l'onglet **Générer**, dans la section **Sortie**. Il faut cocher la case **Fichier de documentation XML** et spécifier le chemin et le nom du fichier généré :

En définissant cette propriété du projet, tout membre qui ne possédera pas de documentation inscrira un avertissement lors de la compilation. Un avertissement n'empêche pas la compilation du projet mais indique qu'une erreur non fatale est présente dans le projet.

Les espaces de noms

Les espaces de noms permettent d'organiser et de hiérarchiser les types. Un espace de noms peut être composé de plusieurs classes écrites dans différents fichiers et compilées dans différentes librairies. C'est le cas du Framework .NET qui regroupe une multitude de types organisés dans des espaces de noms.

1. Le mot clé using

Pour accéder à un type, il suffit de préciser son nom dans la hiérarchie d'espace de noms. Ainsi pour instancier un objet `Form`, il faudra saisir :

```
System.Windows.Forms.Form F = new System.Windows.Forms.Form();
```

Cette notation peut rapidement devenir fastidieuse, c'est pourquoi le mot clé `using` permet de spécifier les espaces de noms utilisés et ainsi réduire le code et augmenter sa lisibilité. Placée en haut du fichier de classe et spécifiant un espace de noms, cette notation permet de n'utiliser que le nom du type :

```
using System.Windows.Forms;
```

L'instanciation se fera de la manière suivante :

```
Form F = new Form();
```

2. Le mot clé alias

En cas de conflit, si le nom d'un type est défini dans deux espaces de noms qui sont tous les deux déclarés avec le mot clé `using`, il faut créer un alias. Prenons pour exemple deux classes de même nom et dans des espaces de noms différents :

```
namespace Class1
{
    class Class
    {
        ...
    }
}
```

et

```
namespace Class2
{
    class Class
    {
        ...
    }
}
```

En déclarant les deux espaces de noms dans l'entête du fichier de classe qui les instancie, le compilateur indique une erreur de référence ambiguë. Il faut donc spécifier un alias pour au moins l'un des deux espaces de noms et l'utiliser pour faire référence au type :

```
using alias = Class1;
using Class2;
...
alias.Class C1 = new alias.Class();
Class C2 = new Class();
```

Les types de base

Les types de données permettent de stocker des valeurs dans l'application. Les langages .NET étant fortement typés, il n'est pas toujours possible de convertir un type de données en un autre. Les conversions, implicites ou explicites, permettent de convertir les types de données. Cela est possible car tous les types du Framework .NET dérivent du type `Object` qui est le type de base de tous les autres types.

1. Les types numériques

Les types numériques sont décomposés en deux parties : les entiers et les décimaux. Chacun dispose d'un ensemble de types pour représenter de la manière la plus judicieuse les données en fonction des besoins.

a. Les entiers

Le tableau suivant répertorie les types d'entiers disponibles dans le Framework .NET:

Type .NET	Nom C#	Description	Plage de valeurs
`System.Byte`	`byte`	Entier non signé de 8 bits	De 0 à 255
`System.Int16`	`short`	Entier signé de 16 bits	De -32 768 à 32 767
`System.Int32`	`int`	Entier signé de 32 bits	De -2 147 483 648 à 2 147 483 647
`System.Int64`	`long`	Entier signé de 64 bits	De -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
`System.SByte`	`sbyte`	Entier signé de 8 bits	De -128 à 127
`System.UInt16`	`ushort`	Entier non signé de 16 bits	De 0 à 65 535
`System.UInt32`	`uint`	Entier non signé de 32 bits	De 0 à 4 294 967 295
`System.UInt64`	`ulong`	Entier non signé de 64 bits	De 0 à 18 446 744 073 709 551 615

Une valeur peut être assignée à un entier avec une notation décimale ou hexadécimale. La notation hexadécimale doit être précédée du préfixe `0x` :

```
int i = 2;              // Notation décimale
i = 0x4B;               // Notation hexadécimale équivalent à: i = 75;
```

b. Les décimaux

Le tableau suivant répertorie les types décimaux disponibles dans le Framework .NET :

Type .NET	Nom C#	Description	Précision
`System.Single`	`float`	Nombre à virgule flottante de 32 bits	7 chiffres significatifs

System.Double	double	Nombre à virgule flottante de 64 bits	15 chiffres significatifs
System.Decimal	decimal	Nombre à virgule flottante de 128 bits	28 chiffres significatifs

2. Les booléens

Un booléen est un type qui permet de représenter une valeur qui est soit true, soit false. Le type .NET correspondant est System.Boolean et son nom C# est bool.

Il est possible d'assigner à un booléen le résultat d'une comparaison :

```
byte x = 1;
bool b = x < 2;        // b à la valeur true
```

3. Les chaînes de caractères

Le type System.String (string) est un type référence qui représente une série de types System.Char (char).

Une variable de type char est assignée avec un caractère placé entre guillemets simples :

```
char c = 'a';
```

Le type char représente une instance d'un caractère Unicode de 16 bits. Il est donc possible d'affecter une valeur à un type char en utilisant la valeur numérique du caractère Unicode qui consiste à fournir un caractère sous forme hexadécimale de 4 chiffres :

```
char c = '\u0061';     // Equivaut à: c = 'a';
```

Une variable de type string est assignée avec une chaîne de caractères placée entre des guillemets doubles :

```
string s = "Ma chaîne";
```

Le type string étant un type référence, il possède des méthodes permettant de manipuler les chaînes. Le tableau suivant présente des méthodes parmi les plus utilisées :

Méthode	Description
Replace	Remplace toutes les occurrences d'un caractère dans la chaîne par un autre.
Split	Sépare la chaîne en plusieurs en fonction d'un caractère délimiteur.
Substring	Retourne une partie de la chaîne.
ToCharArray	Retourne un tableau de type char à partir de la chaîne.
ToLower	Convertit tous les caractères de la chaîne en minuscules.
ToUpper	Convertit tous les caractères de la chaîne en majuscules.
Trim	Supprime les espaces en début et fin de chaîne.

4. Les types nullable

Les types référence peuvent représenter des valeurs non existantes (null) au contraire des types valeur :

```
string s = null;       // Opération autorisée.
```

```
int i = null;          // Erreur de compilation.
```

Pour représenter une valeur `null` dans un type valeur, vous devez utiliser une construction spécifique appelée type nullable. Un type nullable est remarquable grâce au caractère `?` :

```
int? i = null;         // Opération autorisée.
```

Il est ainsi possible de tester si la variable contient une valeur de manière classique ou avec sa propriété `HasValue` :

```
if (i != null)
{ }
```

Équivaut à :

```
if (i.HasValue)
{ }
```

La conversion d'un type classique vers un type nullable est implicite. La valeur est directement affectée à la propriété `Value` du type nullable :

```
int? i = 1;
```

La conversion d'un type nullable vers un type classique doit être explicite. Cela équivaut à affecter la propriété `Value` du type nullable au type classique. Si la propriété `HasValue` est `false` et que la variable est donc `null`, une exception sera levée :

```
int j = (int)i;        // Equivaut à: int j = i.Value;
```

5. La conversion de types

La conversion de types peut être effectuée de deux manières différentes : la conversion implicite qui signifie qu'elle est effectuée de manière automatique lorsqu'il n'y a aucun risque de perte de données et la conversion explicite qui signifie que la conversion et les types doivent être spécifiés.

a. La conversion implicite

Si un type peut être implicitement converti en un autre, il est possible d'utiliser le premier type en remplacement du second sans syntaxe spécifique :

```
int i = 1;
long l = i;
```

Si une méthode doit être appelée avec un paramètre de type `long`, il sera possible de l'appeler en lui passant un paramètre de type `int` sans lever d'erreurs.

Le tableau suivant présente les conversions implicites prises en charge par C# :

Du type	Vers le type
`byte`	short, ushort, int, uint, long, ulong, float, double, decimal
`short`	int, long, float, double, decimal
`int`	long, float, double, decimal
`long`	float, double, decimal
`float`	double
`sbyte`	short, int, long, float, double, decimal
`ushort`	int, uint, long, ulong, float, double, decimal

`uint`	long, ulong, float, double, decimal
`ulong`	float, double, decimal
`char`	int, uint, long, ulong, float, double, decimal

b. La conversion explicite

Quand une conversion concerne des types qui ne peuvent pas être convertis implicitement, il faut les convertir explicitement. Cette conversion s'effectue au moyen d'une syntaxe spéciale :

```
long l = 1;
int i = (int)l;
```

Étant donné que les deux types `int` et `long` peuvent contenir la valeur 1, la conversion s'effectue. Si le type de destination de la conversion ne peut pas contenir la valeur source, la valeur issue de la conversion ne reflètera pas la réalité :

```
short s = 300;
byte b = (byte)s;             // b = 44
```

Le code compile et s'exécute sans lever d'erreurs, il faut donc être très attentif aux conversions explicites qui peuvent être une source d'erreur.

Les tableaux

Les tableaux permettent de grouper des séries de variables et d'y accéder au moyen d'un index basé sur 0. Les tableaux peuvent avoir une ou plusieurs dimensions.

La déclaration d'un tableau se fait en ajoutant `[]` au type de données qui seront stockées et son initialisation se fait en indiquant le nombre maximum d'éléments qu'il pourra contenir :

```
int[] Tab;
Tab = new int[10];
```

La déclaration et l'initialisation peuvent se faire en une seule instruction :

```
int[] Tab = new int[10];
```

Le tableau déclaré dans le code précédent pourra contenir 10 éléments de type `int` et aura un index compris entre 0 et 9.

Il est possible d'initialiser un tableau sans spécifier de limite maximum mais en spécifiant une série de valeurs. Elles sont disposées entre accolades et séparées par des virgules :

```
int[] Tab = new int[] { 1, 2, 5, 9, 12 };
```

Le Framework .NET prend également en charge les tableaux multidimensionnels. Les tableaux rectangulaires sont des tableaux où chacune des lignes possèdent le même nombre de colonnes :

```
int[,] Tab = new int[2, 2];
int[,,] Tab = new int[5, 3, 2];
```

Les tableaux en escaliers sont un autre type de tableaux multidimensionnels. À la différence des tableaux rectangulaires, les tableaux en escaliers ont des lignes qui peuvent avoir un nombre de colonnes différent. Il s'agit en fait d'un tableau de tableaux :

```
int[][] Tab = new int[2][];
Tab[0] = new int[] { 2, 5 };
Tab[1] = new int[] { 2, 5, 12, 21 };
```

Les valeurs des tableaux sont accessibles grâce à l'indexeur qui donne accès directement à la variable en lecture et écriture :

```
int[] Tab = new int[10];
Tab[0] = 1;
for (int i = 1; i < Tab.Length; i++)
{
    Tab[i] = Tab[i - 1] * 2;
}
```

Les collections

Une collection est un type spécialisé qui organise et expose des groupes d'objets. À l'image des tableaux, on accède aux membres par un index. La différence est que les collections sont redimensionnables et qu'il est possible d'ajouter et de supprimer des membres lors de l'exécution.

Les collections se trouvent dans l'espace de noms `System.Collections`. La plus commune des collections est le type `ArrayList` qui permet d'ajouter et de supprimer dynamiquement des éléments soit à la fin, soit à un index prédéterminé :

```
ArrayList maCollection = new ArrayList();
maCollection.Add(new object());
maCollection.Insert(0, "ABCD");
```

La collection de type `ArrayList` contient des objets de type `object`. Il est possible d'insérer plusieurs types d'objet différents. Cela implique que, lors de la récupération, l'objet doit être converti explicitement :

```
string s = (string)maCollection[0];
```

Le principal intérêt des collections est de pouvoir réaliser une boucle de tous les membres grâce à l'instruction `foreach` :

```
foreach (object o in maCollection)
{}
```

Lorsque vous utilisez cette syntaxe, il faut s'assurer que tous les membres ont le même type que la variable d'itération et si ce n'est pas le cas, le corps de la boucle doit comporter un processus pour tester le type de l'objet :

```
foreach (object o in maCollection)
{
    if (o.GetType() == typeof(string))
    {

    }
}
```

Lorsque les objets sont accédés à travers une variable d'itération d'une boucle `foreach`, ils sont en lecture seule. Pour créer une boucle sur une collection afin d'en modifier les membres, il faut utiliser une boucle `for`.

L'espace de noms `System.Collections.Generic` expose des types de collections fortement typées. Lors de la déclaration d'une variable de ce type, il faut spécifier le type des membres qui pourront être stockés. Cela assure que tous les objets d'une collection sont du même type et lors de l'accès à l'un de ses membres, la valeur retournée est du type de la collection :

```
List<string> maCollection = new List<string>();
maCollection.Add("ABCD");
string s = maCollection[0];
foreach (string s in maCollection)
{
}
```

Le type `List` est le plus commun et permet de stocker des objets de la même manière qu'un tableau unidimensionnel. Pour les collections génériques, le type des membres est spécifié entre crochets après le type de la collection.

Les directives Preprocessor

Les directives preprocessor fournissent au compilateur des informations supplémentaires sur les sections du code. Les plus communes de ces directives sont les directives conditionnelles qui permettent d'inclure ou non certaines sections lors de la compilation.

Les directives sont définies dans les propriétés du projet (menu **Projet** - **Propriétés de SelfMailer...**) sous l'onglet **Générer** dans la section **Général** :

Par défaut, Visual Studio génère deux symboles DEBUG et TRACE en mode de compilation Debug et seulement TRACE en mode Release. Il est possible de ne pas générer ces symboles en décochant les cases correspondantes ou d'en ajouter des personnalisés dans le champ **Symboles de compilation conditionnelle**.

L'exemple suivant teste si la constante DEBUG est définie afin d'exécuter le code :

```
class MaClasse
{
    int i;
    public void MaMethode()
    {
#if DEBUG
        MessageBox.Show("i = " + i.ToString());
#endif
    }
}
```

Si la constante DEBUG est définie au moment de la compilation, le code permettant d'afficher une boîte de dialogue avec la valeur de la variable `i` sera compilé.

Le tableau suivant répertorie les directives preprocessor :

Directive preprocessor	Description
`#define symbole`	Définit un nouveau symbole.
`#undef symbole`	Supprime un symbole.
`#if symbole [opérateur symbole2]`	Réalise un test sur l'existence d'un ou plusieurs symboles.
`#else`	Lié à une directive `#if`, le code est compilé si la directive `#if` est fausse.
`#elif symbole [opérateur symbole2]`	Lié à une directive `#if`, le code est compilé si la directive `#if` est fausse et que le test sur le ou les symboles est vrai.
`#endif`	Marque la fin d'une directive `#if`.
`#warning texte`	Spécifie le texte de l'avertissement qui apparaît dans la sortie du compilateur.
`#error texte`	Spécifie le texte de l'erreur qui apparaît dans la sortie du compilateur.
`#region nom`	Marque le début d'une section du code.
`#endregion`	Marque la fin d'une section du code.

Les directives de test telles que `#if` et `#elif` permettent de faire des tests sur l'existence ou non de symboles. Il est possible d'utiliser les opérateurs de comparaison `&&`, `||` et `!` pour faire des tests logiques ET, OU et NON. L'exemple suivant indique au compilateur d'inclure le code si le symbole DEBUG n'est pas défini et que le symbole TRACE est défini :

```
#if !DEBUG && TRACE
    MessageBox.Show("i = " + i.ToString());
#endif
```

L'éditeur de texte colorise en gris les instructions si celles-ci ne sont pas incluses avec la configuration actuelle lors de la compilation. Si les instructions sont incluses lors de la compilation, les instructions sont colorisées de manière classique.

Il faut être conscient que les directives `#if` et `#elif` ne sont pas des expressions C# classiques. Il n'est donc pas possible de faire des tests sur des variables.

Une autre manière de rendre la compilation conditionnelle est d'utiliser l'attribut `Conditional` de l'espace de noms `System.Diagnostics`. Cet attribut permet de définir des méthodes conditionnelles, ainsi si le symbole n'est pas défini, tous les appels aux méthodes marquées sont omis :

```
class MaClasse
{
    int i;
    [Conditional("DEBUG")]
    public void MaMethode()
    {
        MessageBox.Show("i = " + i.ToString());
    }
}
```

L'attribut `Conditional` est applicable avec certaines restrictions :

- Sur une méthode dans une classe ou une structure mais pas sur une méthode d'une interface.
- La méthode ne doit pas avoir de valeur de retour, il ne doit donc pas s'agir d'une fonction.
- La méthode ne doit pas être marquée avec le mot clé `override`, elle peut avoir le mot clé `virtual` mais la surcharge sera implicitement conditionnelle.

Introduction

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

```
class MaClasse
{
}
```

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

- Les membres (méthodes, propriétés, indexeurs, évènements...) sont placés entre les accolades.
- Les attributs et les modificateurs de classe comme le niveau d'accès sont placés avant le mot clé `class`.
- L'héritage et les implémentations d'interfaces sont placés après le nom de la classe.

Les niveaux d'accès

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

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

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

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

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

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

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

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

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

Les structures

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

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

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

La déclaration et l'instanciation d'une occurrence de la structure se font comme pour une classe. Un constructeur sans paramètre existe implicitement mais il ne peut pas être surchargé :

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

Il est possible de rajouter son propre constructeur à partir du moment où il prend des paramètres et que tous les champs de la structure sont initialisés :

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

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

```
struct GeoPoint
{
    double Longitude = 1;     // Opération non autorisée
    double Latitude;
}
```

Les classes

1. Les champs

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

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

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

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

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

```
private string filename = "untitled.smpx", path;
```

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

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

2. Les propriétés

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

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

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

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

Il est possible de créer des propriétés automatiques au lieu de créer un champ puis une propriété avec des

accesseurs qui ont pour seul but de lire et écrire dans un champ privé :

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

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

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

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

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

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

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

L'accesseur `set` est modifié de manière à tester si la valeur du champ `filename` sera modifiée avec la nouvelle valeur. Si oui, la nouvelle valeur est affectée au champ `filename` et la propriété `HasChanged` est modifiée. La propriété `HasChanged` est mise à jour et non directement le champ `hasChanged` de manière à pouvoir effectuer un traitement supplémentaire lors de son affectation comme lever un évènement.

3. Les méthodes

Les méthodes permettent d'exécuter des séries d'instructions. Elles peuvent recevoir des données en tant que paramètres et retourner une valeur. Une méthode spécifiant le mot clé `void`, à la place d'un type de retour, indique qu'aucune donnée ne sera retournée à l'appelant.

La signature d'une méthode est composée de son nom et du type de ses paramètres. Le nom des paramètres n'est pas pris en compte dans la définition de la signature d'une méthode. Cette signature doit être unique au sein d'un type.

Ajoutez la méthode `Save` à la classe `Project`, cette méthode prend deux paramètres booléens :

```
/// <summary>
/// Sauvegarde le projet.
/// </summary>
/// <param name="Ask">Spécifie si l'utilisateur doit confirmer la
/// sauvegarde.</param>
/// <param name="ShowDialog">Spécifie si l'utilisateur a la
/// possibilité de choisir le fichier.</param>
public void Save(bool Ask, bool ShowDialog)
{
    /* Si l'utilisateur doit confirmer la sauvegarde et que le
     * projet a été modifié, l'utilisateur doit cliquer sur le
     * bouton Oui de la boîte de dialogue */
    if (!Ask
      || (this.HasChanged
         && MessageBox.Show(
                "Voulez-vous sauvegarder les modifications ?",
                "Sauver", MessageBoxButtons.YesNo,
                 MessageBoxIcon.Information,
                 MessageBoxDefaultButton.Button1)
         == DialogResult.Yes))
    {
```

```
        /* Si l'utilisateur n'a pas la possibilité de choisir un
         * fichier et que le nom du fichier ainsi que son chemin
         * sont définis, le projet est enregistré et la propriété
         * HasChanged est réinitialisée */
        if (!ShowDialog
          && !string.IsNullOrEmpty(this.Filename)
          && !string.IsNullOrEmpty(this.Path))
        {
            // Sauvegarde des données non implémentées
            this.HasChanged = false;
        }
        /* Dans le cas contraire l'utilisateur doit choisir un
         * chemin et un nom de fichier pour le projet qui sera
         * enregistré */
        else
        {
            /* Instanciation et initialisation de la boîte de
             * dialogue de sauvegarde de fichiers */
            SaveFileDialog sfdProject = new SaveFileDialog();
            sfdProject.Filter = " Self Mailer (*.smpx)|*.smpx";
            sfdProject.RestoreDirectory = true;
            sfdProject.SupportMultiDottedExtensions = true;
            sfdProject.Title = "Sauver le projet";
            /* Affichage de la boîte de dialogue et test de la
             * valeur de retour */
            if (sfdProject.ShowDialog() == DialogResult.OK)
            {
                /* Assignation des valeurs aux variables et
                 * sauvegarde du projet */
                this.Filename =
          System.IO.Path.GetFileName(sfdProject.FileName);
                this.Path =
          System.IO.Path.GetDirectoryName(sfdProject.FileName);
                // Sauvegarde des données non implémentées
                this.HasChanged = false;
            }
        }
    }
}
```

a. La surcharge

Une méthode a la possibilité d'être surchargée. Cela signifie qu'il peut exister plusieurs méthodes avec le même nom tant que leurs signatures sont différentes.

Par exemple, vous pouvez créer les surcharges suivantes à la méthode `Save` de la classe `Project` :

```
public void Save()
{
    this.Save(false, false);
}
public void Save(bool Ask)
{
    this.Save(Ask, false);
}
```

Dans l'exemple ci-dessus les surcharges contiennent des paramètres en moins dans leur signature et le corps des méthodes ne fait qu'appeler la méthode originale avec des paramètres par défaut. L'utilisation des paramètres optionnels, étudiés plus loin dans ce chapitre, sera préférée dans ce cas.

b. Les paramètres

Les méthodes peuvent avoir une séquence de paramètres qui définit les arguments qui doivent être fournis à celles-ci. Il est possible de contrôler la manière dont ces paramètres seront passés à la méthode.

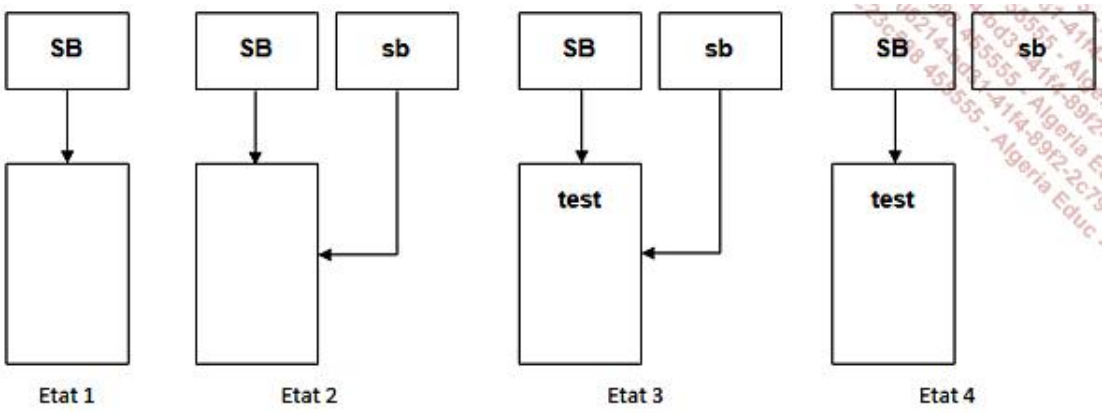
Passage par valeur

Le passage par valeur est le mode de passage par défaut. Cela signifie qu'une copie de la valeur est créée lors du passage à la méthode. La modification de la valeur du paramètre dans le corps de la méthode n'entraînera pas la modification de l'originale.

Pour les types référence, c'est la référence de l'objet qui est copiée et pas l'objet lui-même. Cela signifie que deux variables font référence au même objet en mémoire et qu'une modification affectera l'objet original :

```
static class Program
{
    [STAThread]
    static void Main()
    {
        StringBuilder SB = new StringBuilder();   // Etat 1
        test(SB);                                 // Etat 2
    }
    static void test(StringBuilder sb)
    {
        sb.Append("test");                        // Etat 3
        sb = null;                                // Etat 4
    }
}
```

Dans l'exemple précédent, on crée un objet de type `StringBuilder` qui est un type référence et cet objet est passé à la méthode `test`. La méthode modifie l'objet puis la référence de l'objet est assignée avec la valeur `null`. À la fin de l'exécution de la méthode `test`, l'objet `SB` passé en paramètre à la fonction a bien été modifié avec la nouvelle valeur. Par contre, sa référence n'est pas `null`. La raison est que la modification d'un objet modifie sa valeur en mémoire mais que l'assignation de `null` se fait au niveau du pointeur du paramètre `sb` :



Passage par référence avec ref

Le mot clé `ref` avant le type du paramètre permet de spécifier que le paramètre est passé par référence. Cela signifie qu'aucune copie du type valeur ou référence n'est passée à la fonction, mais c'est la variable originale donc toute modification sur celle-ci est répercutée :

```
static class Program
{
    [STAThread]
    static void Main()
    {
        int I = 0;
        test(ref I);
       // I = 1
    }
    static void test(ref int i)
    {
        i++;
    }
}
```

Remarquez dans l'exemple précédent que le mot clé `ref` est requis non seulement dans la signature de la méthode mais également dans l'appel de la méthode. Si le mot clé est oublié lors de l'appel de la méthode, Visual Studio lèvera une erreur lors de la compilation. Cela permet de signifier clairement ce qui se passe sur la variable.

Passage par référence avec out

Le mot clé `out` est comme `ref`. Il permet de passer un paramètre par référence à l'exception que le paramètre ne doit pas être assigné avant d'être transmis à la méthode d'une part et, que le paramètre doit être assigné avant de quitter la méthode, d'autre part :

```
static class Program
{
    [STAThread]
    static void Main()
    {
        string S1, S2;
        test(out S1, out S2);
       // S1 = "test1"
       // S2 = "test2"
    }
    static void test(out string s1, out string s2)
    {
        s1 = "test1";
        s2 = "test2";
    }
}
```

Le mot clé `params`

Un paramètre marqué avec le mot clé `params` doit toujours être le dernier paramètre de la signature d'une méthode et spécifie que celle-ci accepte n'importe quel nombre de paramètres d'un type particulier. Le paramètre doit également être déclaré en tant que tableau :

```
static class Program
{
    [STAThread]
    static void Main()
    {
        test(1);                                // Retourne 1
        test(1, 2);                             // Retourne 3
        test(new int[] { 1, 2, 4, 8, 16 });     // Retourne 31
    }
    static int test(params int[] I)
    {
        int result = 0;
        for (int i = 0; i < I.Length; i++)
        {
            result += I[i];
        }
        return result;
    }
}
```

Comme dans l'exemple précédent, il est possible de passer un tableau initialisé du type demandé à la place d'une série de valeurs à la méthode :

```
test(new int[] { 1, 2, 4, 8, 16 });
```

Paramètres optionnels

Les paramètres optionnels sont une nouveauté du C# 4.0. Les méthodes, constructeurs et indexeurs peuvent déclarer des paramètres optionnels.

Un paramètre devient optionnel si une valeur par défaut est spécifiée lors de sa déclaration dans la signature de la méthode.

Modifiez la signature de la méthode `Save` de la classe `Project` comme dans l'exemple (et supprimez les méthodes de surcharges) :

```
public void Save(bool Ask = false, bool ShowDialog = false)
```

Les paramètres `Ask` et `ShowDialog` sont devenus optionnels et leur valeur par défaut est `false`. Cela signifie que la méthode `Save` peut être appelée par les instructions suivantes :

```
Project P = new Project();
P.Save();                // Equivalent à P.Save(false, false);
P.Save(true);            // Equivalent à P.Save(true, false);
P.Save(false, true);
```

Vous pouvez remarquer qu'il est impossible de spécifier seulement le second paramètre dans l'appel de la méthode. Les deux paramètres étant du même type, si un seul est spécifié, le compilateur comprendra que c'est le premier paramètre qui est donné et il utilisera le paramètre par défaut pour le second. On est alors obligé de spécifier les deux paramètres.

L'utilisation des paramètres optionnels implique des contraintes :

- Ils ne peuvent pas être marqués avec les mots clés `ref` et `out`.
- Les paramètres obligatoires doivent se situer avant les paramètres optionnels dans la signature de la méthode à l'exception d'un paramètre qui serait marqué avec le mot clé `params` et qui doit toujours rester le dernier dans la signature de la méthode.

Paramètres nommés

Introduits dans la version 4.0 de C#, les paramètres nommés permettent d'identifier les arguments non plus par leur position mais par leur nom. Il devient ainsi possible d'appeler notre méthode `Save` des manières suivantes qui sont équivalentes :

```
P.Save(Ask: true, ShowDialog: true);
P.Save(ShowDialog: true, Ask: true);
```

Il n'est pas nécessaire d'apporter des modifications à la signature de la méthode pour utiliser les paramètres nommés.

Il est possible de mixer les paramètres nommés et les positions :

```
P.Save(true, ShowDialog: true);
```

Par contre, il est interdit de placer des paramètres nommés avant des paramètres de position :

```
P.Save(ShowDialog: true, true);
```

Les paramètres nommés sont particulièrement utiles dans le cadre de méthodes ayant plusieurs paramètres optionnels de même type comme pour l'exemple de la méthode `Save` de la classe `Project`. On pourra ainsi remplacer l'appel de méthode suivant :

```
P.Save(false, true);
```

par celui-ci qui permet de bénéficier de la fonctionnalité du paramètre optionnel et de sa valeur par défaut :

```
P.Save(ShowDialog: true);
```

4. Les constructeurs

Les constructeurs sont des méthodes spécifiques pour les classes et les structures. Ils permettent de fournir le code d'initialisation de l'objet.

La signature d'un constructeur est pratiquement identique à celle d'une méthode à l'exception près qu'elle n'a pas de type de retour et que son nom doit être identique à sa classe.

Créez un constructeur pour la classe `Project` comme suit :

```
public Project()
{
}
```

Comme pour les méthodes classiques, il est possible de surcharger un constructeur. Pour éviter la duplication du code, un constructeur peut en appeler un autre en utilisant le mot clé `this` :

```
public Project()
```

```
    : this("sans titre.smpx")
{
}
public Project(string filename)
{
}
```

Lorsqu'un constructeur en appelle un autre, l'exécution commence par le constructeur appelé puis fini par le constructeur appelant.

Pour les classes, le compilateur C# génère automatiquement un constructeur sans paramètre s'il n'y a aucun autre constructeur de défini. Si la classe possède au moins un constructeur, le constructeur sans paramètre n'est plus généré automatiquement.

Pour les structures, il n'est pas possible de définir un constructeur sans paramètre car le rôle d'une telle méthode est implicitement d'initialiser chaque champ avec sa valeur par défaut.

Lors de l'instanciation d'un objet, avant que le constructeur ne soit exécuté, il y a l'initialisation des champs dans leur ordre de déclaration.

Afin de simplifier l'initialisation des objets, les champs et propriétés accessibles peuvent être initialisés en une seule et unique instruction :

```
Project P = new Project() { HasChanged = true };
```

5. Les destructeurs

Les destructeurs sont des méthodes exécutées automatiquement par le processus de récupération de mémoire pour un objet qui n'est plus référencé. Ces méthodes sont seulement autorisées dans les classes.

La syntaxe d'un destructeur comporte le nom de la classe précédé du caractère ~ :

```
~Project()
{
}
```

Un bon exemple d'utilisation des destructeurs est de fournir une méthode de secours dans les cas où un objet n'aurait pas été détruit via la méthode `Dispose`. Il est préférable d'avoir un objet détruit tardivement plutôt que jamais.

Les destructeurs sont utiles mais il y a quelques bémols :

- Les destructeurs ralentissent l'allocation et la récupération de la mémoire.
- Les objets ont une durée de vie plus longue car ils doivent attendre que le processus de récupération de mémoire les supprime.
- Il est impossible de déterminer à quel moment et dans quel ordre seront exécutés les destructeurs des objets. Le traitement par le processus de récupération de mémoire ne se fait ni sur le modèle de la pile, ni sur le modèle du tas.

6. Les classes et membres statiques

Un membre classique d'une classe est spécifique à une instance d'objet tout comme une classe doit être instanciée. Le mot clé `static` permet de partager des membres entre plusieurs instances et de créer des classes qui n'auront pas besoin d'être instanciées :

```
public class Statique
{
    public static int i;
    public static void test()
    {
    }
}
```

Un membre qui est marqué avec le mot clé `static` existe une seule fois quel que soit le nombre d'instances de la

classe. Le membre partagé est accédé via le type et non pas à partir de l'instance d'objet :

```
Statique.i = 1;
Statique.test();
```

7. Les classes partielles

Le mot clé `partial` autorise une classe, structure, méthode ou interface à être définies dans plusieurs fichiers. En règle générale, une classe est définie dans un seul fichier. Dans certains cas, comme pour les formulaires, Visual Studio gère une partie du code généré automatiquement par le concepteur de vue dans un fichier et laisse au développeur la possibilité d'ajouter la logique dans un autre fichier. Les deux fichiers, ensemble, composent la classe entière. Observez le fichier **Form1.cs** :

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
```

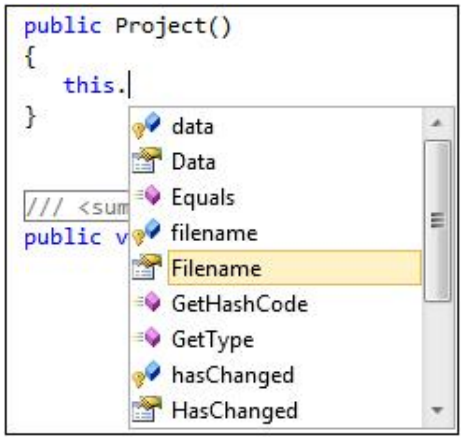
Et le fichier **Form1.Designer.cs** :

```
partial class Form1
{
    private System.ComponentModel.IContainer components = null;

    protected override void Dispose(bool disposing)
    {
        ...
    }

    #region Code généré par le Concepteur Windows Form

    private void InitializeComponent()
    {
        ...
    }

    #endregion
}
```

Le compilateur se charge de fusionner les deux fichiers et il est possible d'appeler les membres de l'un ou de l'autre de la même manière qu'un membre d'une classe normale.

8. Le mot clé this

Le mot clé `this` permet de se référer à l'instance en cours. Ce mot clé permet, par exemple, de lever les ambiguïtés entre une variable locale et un paramètre de méthode. L'éditeur de texte de Visual Studio propose l'IntelliSense pour le mot clé `this` au même titre que s'il s'agissait d'un objet :



Il est interdit d'utiliser ce mot clé pour les membres statiques puisque `this` fait référence à une instance d'objet.

Le mot clé `this` est également utile pour créer des extensions aux méthodes. Dans le cas où vous souhaitez étendre une classe mais qu'il n'y a pas de possibilité d'y accéder, parce que vous n'avez pas le code source ou qu'il n'est pas possible d'en hériter, les méthodes d'extension vont permettre de créer des méthodes statiques qui apparaîtront comme membre de la classe sans en faire réellement partie.

L'exemple suivant crée une méthode d'extension au type `string` :

```
public static class Extension
{
    public static string test(this string s)
    {
        return s.ToLower();
    }
}
```

La classe doit être statique tout comme la méthode qui prend un premier paramètre du type qui doit être étendu (ici `string`) et ce paramètre est précédé du mot clé `this`. C'est ce qui fait que le compilateur considère cette méthode comme une partie du type `string`.

Ainsi, à n'importe quel endroit du code, vous pouvez faire appel à cette méthode comme s'il s'agissait d'un membre du type `string`. Le premier paramètre n'apparaîtra pas dans la signature de la méthode :

```
string s1 = "ABCD";
string s2 = s1.test();
```

Contrairement aux autres méthodes statiques, la méthode est appelée à partir de l'instance de l'objet et non à partir de son type.

Si la classe contient une méthode avec le même nom, la méthode d'extension sera masquée car n'importe quelle méthode d'instance de la classe a la priorité sur les méthodes d'extension.

9. Les indexeurs

Les indexeurs fournissent un moyen intuitif d'accéder aux éléments d'un objet qui encapsule un tableau ou une collection. Les indexeurs sont très similaires aux propriétés mais l'accès se fait par un argument placé entre crochets plutôt que par un nom. Le type `string` contient un indexeur pour accéder aux caractères :

```
string s = "Bonjour";
char c = s[2];                // c = n
```

Pour définir un indexeur, il faut déclarer une propriété nommée `this` et spécifiant un ou plusieurs arguments entre crochets :

```
class Phrase
{
    string[] Mots = new string[] { "Bonjour", "le", "monde" };
    public string this[int position]
    {
        get { return Mots[position]; }
        set { Mots[position] = value; }
```

```
    }
}
```

Vous pouvez ensuite utiliser l'indexeur de cette manière :

```
Phrase p = new Phrase();
string s = p[0];                        // s = "Bonjour"
```

Si un type déclare plusieurs indexeurs, ceux-ci doivent posséder une signature différente :

```
public string this[int position]
{
    get { ... }
    set { ... }
}
public string this[string premiereLettre]
{
    get { ... }
    set { ... }
}
```

Un indexeur peut déclarer plusieurs paramètres de la manière suivante :

```
public string this[int position, premiereLettre]
{
    get { ... }
    set { ... }
}
```

10. La surcharge d'opérateurs

Les exemples de cette section sont disponibles dans les sources sous le projet OperatorOverloading.

Le but de la surcharge d'opérateurs est de pouvoir utiliser les opérateurs d'addition, soustraction, multiplication et autres sur des objets sans avoir à créer des méthodes spécifiques. Prenons l'exemple d'une structure (le comportement est identique pour les classes) représentant un vecteur mathématique :

```
public struct Vector
{
    public int X, Y;

    public Vector(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }

    public override string ToString()
    {
        return string.Format("X={0} Y={1}", this.X, this.Y);
    }
}
```

a. Les opérateurs arithmétiques

Pour réaliser une addition entre deux vecteurs, vous pouvez créer une méthode `Addition` prenant en paramètres la composante `X` et la composante `Y` du vecteur afin de réaliser l'opération :

```
public Vector Addition(Vector v)
{
    return new Vector()
    {
        X = this.X + v.X,
```

```
        Y = this.Y + v.Y
    };
}
```

Pour additionner deux vecteurs, vous utiliseriez alors une instruction de ce type :

```
Vector A = new Vector(1, 2);
Vector B = new Vector(2, 4);
Vector C = A.Addition(B);
```

La surcharge d'opérateur va permettre de rendre plus intuitive l'utilisation de l'objet avec une notation du type :

```
Vector C = A + B;
```

La syntaxe de la signature d'une surcharge d'opérateur est la suivante :

```
public static type operator signe (Arg1[,Arg2])
```

Dans cette signature, le type représente le type sur lequel l'opérateur agira et le type de l'objet retourné. Le signe représente l'opérateur qui sera surchargé, quant aux arguments il y en aura un seul dans le cas d'une opération unaire et il sera du même type que le type géré par la surcharge. En cas de surcharge d'un opérateur binaire, il y aura deux arguments et l'un d'entre eux, au moins, devra être identique au type géré par la surcharge. L'opérateur surchargé doit obligatoirement être public et statique.

La surcharge de l'opérateur d'addition pour la structure `Vector` donnera :

```
public static Vector operator +(Vector v1, Vector v2)
{
    return new Vector()
    {
        X = v1.X + v2.X,
        Y = v1.Y + v2.Y
    };
}
```

Le corps de la méthode n'est pas différent de celui de la méthode `Addition` défini plus tôt, par contre c'est son utilisation qui est complètement différente :

```
Vector D = A + B + C;
```

Vous pouvez créer plusieurs surcharges pour un même opérateur à partir du moment où leurs signatures sont différentes :

```
public static Vector operator +(Vector v1, int i)
{
    return new Vector()
    {
        X = v1.X + i,
        Y = v1.Y + i
    };
}
```

Vous pouvez ainsi réaliser une opération d'addition entre un objet `Vector` et un entier :

```
Vector E = A + 10;
```

La surcharge d'un opérateur arithmétique inclut également celle de son opérateur d'assignement. Vous pouvez donc utiliser la notation suivante :

```
A += 10;
```

La surcharge d'opérateurs fonctionne sur le même modèle pour tous les opérateurs arithmétiques (`+`, `-`, `*`, `/`, `%`).

b. Les opérateurs de comparaison

Il existe six opérateurs de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`. Vous pouvez aussi les surcharger au même titre que les opérateurs arithmétiques avec quelques points spécifiques :

- La surcharge des opérateurs de comparaison doit être effectuée par paires. Si vous surchargez l'opérateur ==, vous devrez aussi surcharger != sinon une erreur sera levée au moment de la compilation. Les paires sont les suivantes :
    - == et !=
    - < et >
    - <= et >=
- Les opérateurs de comparaison doivent obligatoirement retourner un type `bool`. Cela n'aurait pas de sens autrement.
- Si vous surchargez les opérateurs == et !=, vous devez aussi fournir des surcharges pour les méthodes `Equals` et `GetHashCode` héritées du type `Object` sinon des avertissements lors de la compilation seront levés. La raison principale est que la méthode `Equals` doit utiliser la même implémentation logique que l'opérateur ==.

Voici l'exemple de la surcharge des opérateurs == et != pour la structure `Vector` :

```
public static bool operator ==(Vector v1, Vector v2)
{
    return v1.X == v2.X && v1.Y == v2.Y;
}

public static bool operator !=(Vector v1, Vector v2)
{
    return !(v1 == v2);
}

public override bool Equals(object obj)
{
    if (obj is Vector)
    {
        return this == (Vector)obj;
    }
    else
    {
        return false;
    }
}

public override int GetHashCode()
{
    return base.GetHashCode();
}
```

La surcharge de l'opérateur == effectue une comparaison des variables `X` et `Y` entre elles pour déterminer si les deux objets sont identiques. La surcharge de l'opérateur != renvoie l'inverse du résultat de la surcharge de l'opérateur == entre les deux arguments `v1` et `v2`.

La surcharge de la méthode `Equals` utilise la surcharge de l'opérateur == pour déterminer le résultat retourné. À noter le test sur le type de l'argument qui est de type `object` pour déterminer s'il est du type `Vector`. Si `obj` n'est pas de type `Vector`, il ne peut pas être égal à l'instance de l'objet `Vector` sur laquelle est appelée la méthode `Equals`.

La surcharge de la méthode `GetHashCode` ne fait que renvoyer le résultat de la méthode `GetHashCode` de la classe de base.

Voici un exemple d'utilisation de ces surcharges :

```
Vector A = new Vector(11, 12);
Vector B = new Vector(11, 12);
```

```
Console.WriteLine("A == B : {0}", A == B);
Console.WriteLine("A != B : {0}", A != B);
```

```
Console.WriteLine("A.Equals(B) : {0}", A.Equals(B));
Console.WriteLine("A.Equals(new object()) : {0}",
                                    A.Equals(new object()));
```

La console affiche le résultat suivant :

```
A == B : True
A != B : False
A.Equals(B) : True
A.Equals(new object()) : False
```

L'héritage de classe

Comme vu précédemment, toutes les classes du Framework .NET dérivent de la classe `System.Object`. L'héritage peut-être mis en œuvre de deux manières, l'héritage de classe et l'héritage d'interfaces.

L'héritage implique qu'un type dérive d'un type de base en prenant tous ses membres accessibles. L'héritage est utile lorsque vous devez ajouter des fonctionnalités à un type existant ou quand plusieurs types partagent les mêmes fonctionnalités.

1. Implémenter l'héritage

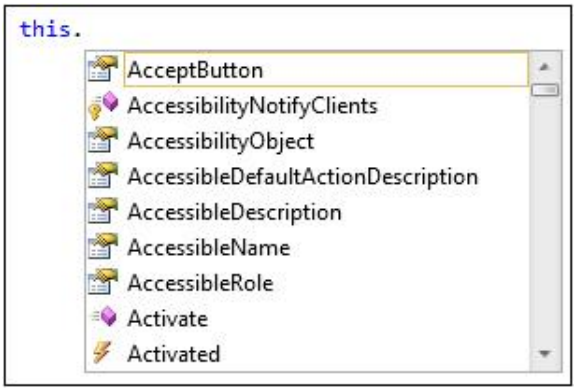
La déclaration de l'héritage utilise une syntaxe simple, il suffit de rajouter le nom du type de base après le nom de la classe dérivée et le caractère :. Une classe ne peut dériver que d'un seul et unique type de base.

Le meilleur exemple est le formulaire **Form1.cs**.

Ouvrez le fichier dans l'éditeur de texte et observez la déclaration de la classe :

```
public partial class Form1 : Form
```

La classe `Form1` dérive de la classe de base `Form`.`Form1` contient donc tous les membres accessibles de sa classe de base. L'IntelliSense permet de les retrouver facilement :



Si aucune classe de base n'est spécifiée, le compilateur considère que `System.Object` est la classe de base. Ainsi, les deux exemples ci-dessous sont identiques :

```
public class Project : System.Object        // Héritage explicite
{
}
public class Project                        // Héritage implicite
{
}
```

Pour la simplicité, l'héritage de la classe `Object` n'est jamais explicitement marqué dans la définition d'une classe.

Les membres qui sont accessibles depuis la classe dérivée sont ceux qui ont un niveau d'accès `public`, `protected` ou `internal`. Les membres privés (`private`) ne seront pas visibles par la classe héritée.

2. Les membres virtuels

En déclarant un membre de la classe de base avec le mot clé `virtual`, vous autorisez le membre à être surchargé par les classes dérivées. Cela s'applique aux méthodes et aux propriétés :

```
public class ClasseDeBase
{
    public virtual bool ProprieteDeBase { get; set; }
    public virtual void MethodeDeBase()
    {
    }
```

```
}
```

Lorsqu'une méthode virtuelle est surchargée dans une classe dérivée, l'appel de la méthode entraîne l'exécution de la méthode surchargée et la méthode de base ne sera pas appelée. La méthode surchargée doit être explicitement déclarée en utilisant le mot clé `override` :

```
public class ClasseEnfant : ClasseDeBase
{
    public override void MethodeDeBase()
    {
    }
}
```

La signature de la méthode dérivée doit correspondre à la signature de la méthode de base. Dans le cas contraire, le compilateur lèvera une erreur car il verra une méthode avec le mot clé `override` mais ne trouvera aucune méthode correspondante dans la classe de base.

3. Masquer les membres hérités

Si une méthode de la classe enfant possède la même signature qu'une méthode de la classe de base et qu'elles ne sont déclarées ni avec le mot clé `virtual`, ni avec `override`, alors on dit que la méthode enfant masque la méthode de base :

```
public class ClasseDeBase
{
    public virtual bool ProprieteDeBase { get; set; }
    public void MethodeDeBase()
    {
    }
}
public class ClasseEnfant : ClasseDeBase
{
    public void MethodeDeBase()
    {
    }
}
```

Au moment de la compilation, un avertissement est levé indiquant au développeur que la méthode masque le membre de la classe de base. Cet avertissement permet de clarifier le fait que la méthode de base ne sera pas appelée et que si c'est un choix intentionnel, il faut ajouter le mot clé `new` sur la méthode enfant :

```
public new void MethodeDeBase()
```

L'avertissement est également déclenché si la méthode de base est marquée comme étant virtuelle, mais le compilateur conseille d'ajouter le mot clé `override` plutôt que `new`.

4. Le mot clé base

Lorsqu'un membre est surchargé, le membre de base n'est plus accessible depuis une instance de la classe mais il reste accessible depuis la classe héritée grâce à la syntaxe `base.[MethodeDeBase]()`:

```
public override void MethodeDeBase()
{
    base.MethodeDeBase();
    bool b = base.ProprieteDeBase;
}
```

Vous pouvez utiliser cette syntaxe pour appeler n'importe quel membre de la classe de base à partir de n'importe quelle portion de code de la classe enfant.

5. Les classes et membres abstraits

Le langage C# autorise les classes et les membres à être marqués avec le mot clé `abstract`. Une classe abstraite ne

peut pas être instanciée et une méthode abstraite ne contient pas d'implémentation, seulement sa signature. Cela implique qu'une classe abstraite ne peut être utilisée que dans le cadre d'un héritage et que cette classe dérivée doit implémenter les fonctionnalités des membres :

```
    public abstract class ClasseDeBase
    {
        public abstract bool ProprieteAbstraite { get; set; }
        public abstract void MethodeAbstraite();
    }
```

Une classe abstraite a pour but de définir la forme des classes enfants sans en définir le fond. L'implémentation finale est à la charge de la classe enfant.

À partir du moment où une classe contient un membre abstrait, celle-ci devient également abstraite et doit être marquée comme telle avec le mot clé `abstract`. Une classe abstraite peut néanmoins contenir des membres non abstraits qui implémentent leur propre logique. Ces membres non abstraits peuvent également être marqués comme virtuels afin d'être surchargés :

```
public abstract class ClasseDeBase
{
    public virtual bool ProprieteDeBase { get; set; }
    public abstract bool ProprieteAbstraite { get; set; }
    public virtual void MethodeDeBase()
    {
    }
    public abstract void MethodeAbstraite();
}
```

La classe qui hérite d'une classe abstraite doit implémenter tous les membres abstraits avec le mot clé `override` :

```
public class ClasseEnfant : ClasseDeBase
{
    public override bool ProprieteAbstraite { get; set; }
    public override void MethodeAbstraite()
    {
    }
}
```

Il est également possible de créer une classe enfant abstraite qui dérive d'une classe abstraite. Il n'est alors pas nécessaire d'implémenter tous les membres de la classe parente :

```
public abstract class ClasseEnfant : ClasseDeBase
{
}
```

6. Les classes et les méthodes scellées

Le mot clé `sealed` sur une déclaration de classe permet d'interdire qu'une autre classe en hérite. Par définition, une classe scellée ne pourra jamais être abstraite puisque ces deux modificateurs sont naturellement opposés. Au même titre, les membres ne peuvent pas être virtuels ni posséder le modificateur d'accès `protected` :

```
public sealed class ClasseDeBase
{
    public bool ProprieteDeBase { get; set; }
    public void MethodeDeBase()
    {
    }
}
```

Une classe peut ne pas être scellée et contenir des membres qui le sont. Ces membres ne pourront donc pas être surchargés par les classes enfants. Pour affecter le modificateur `sealed` à un membre, il faut qu'il soit hérité d'une classe de base :

```
public class ClasseDeBase
{
    public virtual void MethodeDeBase()
    { }
```

```
}
public class ClasseEnfant : ClasseDeBase
{
    public sealed override void MethodeDeBase()
    { }
}
public class ClasseEnfant2 : ClasseEnfant
{
    public override void MethodeDeBase() // Non autorisé
    { }
}
```

Cela permet de stopper l'héritage du membre et de prévenir toute surcharge à partir de classes enfants, bien qu'il soit toujours possible de réaliser une surcharge avec le mot clé `new` :

```
public class ClasseEnfant2 : ClasseEnfant
{
    public new void MethodeDeBase() // Autorisé
    { }
}
```

7. Les constructeurs dérivés

Une classe enfant doit implémenter ses propres constructeurs. Les constructeurs de la classe de base ne sont pas directement accessibles hors de la classe enfant. Les constructeurs par défaut (sans paramètre) sont toujours présents et le constructeur enfant qui implémente son propre constructeur sans paramètre ne doit pas faire explicitement référence au constructeur parent pour que les deux constructeurs soient exécutés en commençant par celui du type parent.

L'exemple suivant déclare deux classes. La classe de base contient une implémentation spécifique pour le constructeur sans paramètre et la classe enfant ne contient aucun constructeur :

```
public class ClasseDeBase
{
    public virtual int ProprieteDeBase { get; set; }
    public ClasseDeBase()
    {
        this.ProprieteDeBase = 10;
    }
    public ClasseDeBase(int i)
    {
        this.ProprieteDeBase = i;
    }
}
public class ClasseEnfant : ClasseDeBase
{
}
```

L'instanciation d'un nouvel objet ne pourra se faire qu'avec un constructeur sans paramètre :

```
ClasseEnfant ce = new ClasseEnfant(true);   // Non autorisé
ClasseEnfant ce = new ClasseEnfant();       // Autorisé
```

Comme la classe enfant ne contient pas d'implémentation explicite pour le constructeur sans paramètre, c'est le constructeur sans paramètre du parent qui sera exécuté.

Si la classe enfant implémente son propre constructeur par défaut, l'instanciation d'un objet enfant entraînera implicitement l'exécution du constructeur parent puis celle du constructeur enfant :

```
public ClasseEnfant()
{
    this.ProprieteDeBase = 5;
}
```

Après l'instanciation, le membre `ProprieteDeBase` a la valeur 5 :

```
ClasseEnfant ce = new ClasseEnfant();
```

Le mot clé base permet de spécifier explicitement quel constructeur de la classe parent il faut exécuter et ainsi de modifier ce comportement :

```
public ClasseEnfant()
    : base(5)
{ }
```

Le mot clé base peut-être utilisé pour tous les constructeurs de la classe enfant, qu'ils aient ou non des paramètres afin de faire référence à n'importe quel constructeur de la classe parente. Les paramètres peuvent être des valeurs comme dans l'exemple précédent ou être des paramètres du constructeur enfant comme dans l'exemple suivant :

```
public ClasseEnfant(int i)
    : base(i)
{ }
```

Quelle que soit la notation choisie, le constructeur de la classe parente sera toujours appelé en choisissant par défaut le constructeur sans paramètre si aucun autre n'est explicitement spécifié. De plus, les constructeurs au sein d'un héritage seront toujours exécutés du plus lointain ancêtre jusqu'à l'objet instancié. Cela permet d'assurer la bonne initialisation des parents.

8. Le polymorphisme

L'héritage entraîne le polymorphisme des classes. Cela signifie que toute classe dérivée peut implicitement être convertie en un objet de sa classe de base. Prenons l'exemple de ces deux classes :

```
public class ClasseDeBase
{
    public int ProprieteDeBase { get; set; }
}
public class ClasseEnfant : ClasseDeBase
{
    public int ProprieteEnfant { get; set; }
}
```

Il est possible de déclarer et d'instancier un objet du type ClasseEnfant puis de l'affecter à un objet de type ClasseDeBase :

```
ClasseEnfant ce = new ClasseEnfant();
ClasseDeBase cb = ce;
```

Les membres spécifiques de la classe enfant ne sont alors plus disponibles dans l'objet qui est du type parent :

```
int i = cb.ProprieteEnfant;        // Non autorisé
```

L'objet n'est pas altéré par la conversion vers un type parent. Les données spécifiques au type enfant sont toujours présentes en mémoire mais elles ne peuvent simplement pas être exploitées.

La conversion inverse, du type parent vers le type enfant, doit être faite de manière explicite :

```
ce = (ClasseEnfant)cb;
```

Si la conversion explicite échoue, parce que l'objet source ne peut pas être converti dans le type de destination, une exception du type InvalidCastException sera levée au moment de l'exécution. Pour éviter de lever une exception à l'exécution, la conversion peut se faire avec l'opérateur as. L'objet destination aura alors une référence null en cas d'échec :

```
ce = cb as ClasseEnfant;
```

Il faut ensuite tester si le résultat de la conversion n'est pas null pour utiliser l'objet destination sans lever d'erreur :

```
if (ce != null)
{
    // La conversion a réussie
}
```

Sans ce test, la conversion sans l'opérateur as est plus avantageuse car l'erreur qui est levée est plus descriptive :

```
int i = ((ClasseEnfant)cb).ProprieteEnfant;
int i = (cb as ClasseEnfant).ProprieteEnfant;
```

En cas d'échec, la première conversion lèvera une erreur du type `InvalidCastException` qui indique précisément que c'est la conversion qui a échouée et non que la variable `cb` est `null`. La seconde conversion, en cas d'échec, lèvera une exception du type `NullReferenceException`. Cela peut vouloir dire d'une part, que la variable `cb` n'est pas du type `ClasseEnfant` et que la conversion a donné un résultat `null`, donc l'appel du membre `ProprieteEnfant` lève l'exception ou d'autre part cela peut vouloir dire que la variable `cb` est `null`.

L'opérateur `is` permet de tester le type d'un objet et par conséquent de prédire le résultat d'une conversion :

```
ClasseEnfant ce = new ClasseEnfant();
ClasseDeBase cb = ce;
if (cb is ClasseDeBase)              // Vrai
{ }
if (cb is ClasseEnfant)              // Vrai
{ }
```

Même si l'objet est converti dans son type parent, le test avec l'opérateur `is` reflète que l'objet est à l'origine du type enfant. Ce qui n'est pas vrai si l'objet est déclaré comme un type parent :

```
ClasseDeBase cb = new ClasseDeBase();
if (cb2 is ClasseDeBase)             // Vrai
{ }
if (cb2 is ClasseEnfant)             // Faux
{ }
```

Les interfaces

Les interfaces vont permettre de définir des comportements pour les classes qui les implémentent. Les classes qui vont implémenter la même interface seront capables d'interagir de façon polymorphe. Il en résulte que tout type implémentant une interface fournit obligatoirement une implémentation des membres définis dans cette interface.

L'interface ne fait que définir les membres qu'un type devra implémenter de la même manière qu'une classe abstraite (à la différence que les types ne dériveront pas de cette classe abstraite et pourront donc être totalement indépendants les uns des autres). Par contre comme pour les classes abstraites, une interface ne peut pas être instanciée.

1. L'implémentation d'interfaces

La syntaxe de déclaration d'une interface est proche de la déclaration d'une classe en utilisant le mot clé `interface`.

Dans le projet **SelfMailer**, créez un nouvel élément dans le dossier **Library** (**Ctrl** + **Maj** + **A** lorsque le dossier est sélectionné). Dans la fenêtre **Ajouter un nouvel élément**, sélectionnez **Interface** et nommez le fichier **IReportChange.cs**. Par convention les noms des interfaces commencent traditionnellement par la lettre i majuscule de manière à pouvoir les repérer plus facilement dans le code. Mais rien n'empêche de définir un nom qui ne commence pas par ce caractère. Cette interface sera implémentée sur les classes du projet afin que les classes implémentent un système de rapport de leurs changements. L'interface se présente de la manière suivante :

```
interface IReportChange
{
}
```

Définissez les membres de l'interface :

```
interface IReportChange
{
    bool HasChanged
    {
        get;
        set;
    }
    event EventHandler Changed;
}
```

Les membres de l'interface ne doivent pas déclarer des modificateurs d'accès car les membres d'une interface sont implicitement publics. Il est également interdit de spécifier que les membres sont virtuels ou statiques. Ces implémentations sont faites au niveau de la classe qui dérivera de l'interface.

L'implémentation des membres dans une classe doit comporter l'exacte signature des membres de l'interface. Si une classe dérive d'une interface mais n'implémente pas ses membres, le compilateur lèvera une exception. Faites dériver la classe `Project` de l'interface `IReportChange` :

```
public class Project : IReportChange
```

Une classe peut dériver de plusieurs interfaces. Les noms des interfaces sont alors séparés par des virgules.

Si vous essayez de compiler (**F5**), le compilateur lèvera une exception indiquant que la classe `Project` n'implémente pas le membre `Changed` de l'interface `IReportChange`. Ajoutez donc ce membre à la classe `Project` :

```
public event EventHandler Changed;
```

Pour illustrer la manière dont plusieurs classes peuvent implémenter la même interface, créez une nouvelle classe dans un fichier **MailServerSettings.cs** dans le dossier **Library**. Faites dériver cette classe de l'interface `IReportChange` et ajoutez l'implémentation des membres :

```
public class MailServerSettings : IReportChange
{
    protected bool hasChanged;

    public bool HasChanged
    {
        get { return hasChanged; }
        set { hasChanged = value; }
```

```
    }

    public event EventHandler Changed;
}
```

2. Le polymorphisme d'interface

Le polymorphisme d'interface implique que tout objet qui implémente une interface est capable de pouvoir être converti dans le type de cette interface et les membres implémentés peuvent être appelés à partir de cet objet converti.

Pour illustrer le polymorphisme des interfaces, modifiez l'accesseur `set` de la propriété `HasChanged` de la classe `MailServerSettings` afin que tout changement déclenche l'évènement `Changed` :

```
public bool HasChanged
{
    get { return hasChanged; }
    set
    {
        if (this.hasChanged != value)
        {
            this.hasChanged = value;
            if (this.Changed != null)
                this.Changed(this, new EventArgs());
        }
     }
}
```

Dès que la variable `hasChanged` d'un objet de type `MailServerSettings` sera modifiée, l'évènement `Changed` sera levé et tout objet abonné à l'évènement pourra réagir en conséquence.

Créez une classe nommée `ProjectSettings` qui sera l'exacte copie de la classe `MailServerSettings`.

Ajoutez un premier membre de type `MailServerSettings` et un second de type `ProjectSettings` à la classe `Project` :

```
public ProjectSettings ProjectSettings { get; set; }
public MailServerSettings MailServerSettings { get; set; }
```

Initialisez ces variables dans le constructeur de la classe `Project` et associez la méthode `ChildChanged` aux évènements `Changed` :

```
public Project()
{
    this.ProjectSettings = new ProjectSettings();
    this.ProjectSettings.Changed +=
                            new EventHandler(ChildChanged);
    this.MailServerSettings = new MailServerSettings();
    this.MailServerSettings.Changed +=
                            new EventHandler(ChildChanged);
}
```

Ajoutez l'implémentation de la méthode `ChildChanged` suivante :

```
public void ChildChanged(object sender, EventArgs e)
{
    if (sender is IReportChange)
    {
        IReportChange Child = (IReportChange)sender;
        this.HasChanged = Child.HasChanged;
        MessageBox.Show("Changement de l'objet de type " +
                                    Child.GetType().ToString());
    }
}
```

Finissez par ajouter les instructions suivantes dans la méthode `Main` de la classe `Program` :

```
Library.Project P = new Library.Project();
```

```
P.MailServerSettings.HasChanged = true;
P.ProjectSettings.HasChanged = true;
```

En lançant l'application (**F5**), l'évènement `Changed` est levé avec comme déclencheur un objet de type `MailServerSettings` puis avec un objet de type `ProjectSettings`. Les deux peuvent être convertis en type `IReportChange` et les membres qu'ils implémentent sont disponibles à partir du résultat de cette conversion. La boîte de dialogue qui affiche le type de l'objet `Child` indique son type d'origine et non le type `IReportChange` de la même façon que pour les classes héritées. Cela signifie qu'un objet du type d'une interface peut faire référence à n'importe quelle instance d'objet dont le type implémente cette interface.

3. L'héritage d'interfaces

Comme les classes, les interfaces peuvent hériter d'une autre interface. Créez une nouvelle interface nommée `IReportChildrenChange` et faites-la hériter de l'interface `IReportChange`. Créez le membre `ChildChanged` de l'interface `IReportChange` vers l'interface `IReportChildrenChange`. Vous devriez obtenir l'interface suivante :

```
interface IReportChildrenChange
{
    void ChildChanged(object sender, EventArgs e);
}
```

Comme l'interface `IReportChildrenChange` hérite de l'interface `IReportChange`, elle contient tous les membres de l'interface parente.

Cette nouvelle interface est destinée aux classes qui implémentent l'interface `IReportChange` et qui ont des variables qui l'implémentent aussi. Elle oblige l'ajout de la méthode `ChildChanged` pour traiter les évènements `Changed` de ses membres. La classe `Project` doit donc être modifiée pour implémenter cette interface :

```
public class Project : IReportChildrenChange
```

Introduction

Les types génériques permettent de combiner la réutilisabilité du code et la sécurité du type. Les types génériques sont le plus souvent employés dans le cadre de collections. Le Framework .NET expose ces collections dans l'espace de noms `System.Collections.Generic` comme le type `List<T>` ou `Dictionary<TKey, TValue>`. Bien entendu, il est possible de créer ses propres types génériques afin de fournir une solution adaptée.

L'avantage des types génériques par rapport aux collections classiques comme le type `ArrayList` est que le type des objets est conservé alors qu'une collection non générique stocke les données en faisant une conversion en type `Object`. La récupération d'un élément d'une collection classique oblige à faire la conversion inverse pour correspondre avec le type attendu tandis que les types génériques retournent un objet déjà typé. Malgré la complexité de codage légèrement supérieure, les types génériques apportent, en plus de la sureté, beaucoup plus de rapidité surtout lorsque les éléments de la liste sont des types valeur.

La création de types génériques

Un type générique est défini dans la déclaration de la classe en plaçant le paramètre `T`. Ce paramètre spécifie que le type sera choisi par le consommateur de la classe. Cela peut être un type valeur ou un type référence.

Créez une nouvelle classe `ReportChangeList<T>` dans le dossier **Library** du projet :

```
public class ReportChangeList<T>
{
}
```

Le paramètre `T` accepte un type qui sera spécifié au moment de l'instanciation :

```
ReportChangeList<int> Ex1 = new ReportChangeList<int>();
ReportChangeList<Object> Ex2 = new ReportChangeList<Object>();
```

Un type générique peut également faire référence à plusieurs classes :

```
public class ClasseGenerique<T, U>
```

Les types génériques peuvent être surchargés tant que le nombre de leurs paramètres de type n'est pas identique :

```
public class ClasseGenerique<T>
public class ClasseGenerique<T, U>
```

Si deux types génériques ont le même nom et le même nombre de paramètres de type, le compilateur lèvera une erreur :

```
public class ClasseGenerique<T>
public class ClasseGenerique<U>         // Non autorisé
```

La classe peut contenir des membres qui vont utiliser le type spécifié à l'instanciation grâce au paramètre `T`. Ajoutez le membre `children` de type `List<T>` à la classe :

```
protected List<T> children;
```

Comme `List<T>` est un type référence, il doit être instancié dans le constructeur pour ne pas être `null`.

Ajoutez le constructeur qui instancie la variable `children` :

```
public ReportChangeList()
{
    this.children = new List<T>();
}
```

Lors de l'instanciation, la classe `ReportChangeList` instanciera une liste générique avec le type qui aura été spécifié.

Le but principal de créer un type générique contenant une liste générique est de pouvoir brider les fonctionnalités exposées ou d'en ajouter. La classe `ReportChangeList` contiendra des enfants dans sa liste `children` et devra notifier l'objet contenant d'un changement. Ajoutez l'interface `IReportChildrenChange` dans la déclaration de la classe :

```
public class ReportChangeList<T> : IReportChildrenChange
```

Puisque la classe hérite de l'interface `IReportChildrenChange`, il faut ajouter les membres définis dans cette interface :

```
protected bool hasChanged;

public bool HasChanged
{
    get
    {
        bool result = false;
        foreach (IReportChange child in this.children)
            if (child.HasChanged)
            {
                result = true;
                break;
            }
```

```
        return this.hasChanged || result;
    }
    set
    {
        if (this.HasChanged != value)
        {
            this.hasChanged = value;
            if (this.Changed != null)
                this.Changed(this, new EventArgs());
        }
        if (!value)
        {
            foreach (IReportChange child in this.children)
                child.HasChanged = value;
        }
    }
}

public event EventHandler Changed;

public void ChildChanged(object sender, EventArgs e)
{
    if (this.Changed != null)
        this.Changed(sender, e);
}
```

Dans le code précédent, les accesseurs de la propriété `HasChanged` réalisent une boucle sur les éléments de la liste `children` afin de faire la mise à jour des objets pour l'accesseur `set`, ou de déterminer si l'objet lui-même ou l'un des éléments de la liste a été modifié pour l'accesseur `get`.

Pour déterminer si l'objet a été modifié, le principe est que si au moins un élément de la liste a été modifié ou l'objet lui même, l'objet entier est considéré comme ayant été modifié. La mise à jour de la propriété `HasChanged` des éléments de la liste ne se fait que si l'objet reçoit la valeur `false` pour sa propriété `HasChanged` car l'objet peut être modifié mais pas ses enfants alors que si les enfants sont modifiés, l'objet parent est par conséquent modifié puisque les enfants font partie de l'objet.

Les contraintes de type

Comme vu dans les accesseurs, l'interface `IReportChange` est utilisée pour accéder à la propriété `HasChanged` des éléments de la liste. Cela signifie que les éléments ajoutés doivent obligatoirement implémenter l'interface. Pour limiter les types qui pourront se substituer au paramètre générique `T`, des contraintes peuvent être appliquées. Voici les contraintes existantes :

Contrainte	Description
`where T :` *classe*	Contrainte sur la classe de base du type
`where T :` *interface*	Contrainte sur l'interface implémentée par le type
`where T : class`	Le type doit être un type référence
`where T : struct`	Le type doit être une structure
`where T : new()`	Le type doit avoir un constructeur sans paramètre
`where U : T`	Le type représenté par `U` doit être identique au type `T`

Ajoutez une contrainte sur l'interface `IReportChange` sur la classe générique `ReportChangeList` :

```
public class ReportChangeList<T> : IReportChildrenChange where T
: IReportChange
```

Les contraintes sont appliquées sur tous les paramètres de type définis, que ce soit dans les méthodes ou dans la définition de type.

Les contraintes de type peuvent également être définies au niveau des méthodes :

```
public void MaMethode<T>() where T : class
{ }
```

Les interfaces génériques

Comme pour les listes, il serait intéressant de pouvoir effectuer une boucle `foreach` sur les éléments de notre type générique. Pour cela il suffit d'implémenter l'interface générique `IEnumerable<T>` :

```
public class ReportChangeList<T> : IReportChildrenChange,
IEnumerable<T> where T : IReportChange
```

Cette interface générique se comporte comme une interface classique, elle contient les signatures des membres requis pour son implémentation et ceux-ci peuvent utiliser la classe générique comme référence.

Ajoutez les membres `GetEnumerator()` et `IEnumerable.GetEnumerator()` suivant pour implémenter l'interface dans la classe générique `ReportChangeList` :

```
public IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < this.children.Count; i++)
    {
        yield return this.children[i];
    }
}
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

L'implémentation d'un énumérateur consiste à effectuer une boucle sur les éléments de la liste et à retourner chacun d'eux à l'appelant. Le mot clé `yield return` permet de faire un retour à l'appelant avec la valeur à retourner en fonction de la précédente valeur retournée. L'état de la méthode est maintenu de telle manière qu'elle peut continuer son exécution au prochain appel. La durée de vie de cet état est liée à l'énumérateur, l'état de la méthode est supprimé lorsque l'énumération est terminée.

1. La variance dans les interfaces génériques

La covariance et la contravariance sont des concepts déjà utilisés depuis la version 2.0 du Framework .NET. La nouveauté avec le Framework .NET 4.0 est de pouvoir appliquer ces principes aux interfaces génériques.

a. La covariance

Une interface covariante autorise ses membres à retourner des types plus dérivés que ceux qui sont spécifiés. L'interface `IEnumerable<T>` implémentée dans la classe `ReportChangeList` fait partie des interfaces covariantes. Cela permet de réutiliser des méthodes fonctionnant avec les collections génériques de base pour les collections dérivées. L'exemple suivant illustre la covariance :

```
public class Classe1
{
    public string Prop1 { get; set; }
    public string Prop2 { get; set; }
}
public class Classe2 : Classe1 { }

static class Program
{
    public static void Write(IEnumerable<Classe1> Elements1)
    {
        foreach (Classe1 elem in Elements1)
        {
            Console.WriteLine("{0} {1}", elem.Prop1, elem.Prop2);
        }
    }

    [STAThread]
    static void Main()
    {
        IEnumerable<Classe2> Elements2 = new List<Classe2>();
```

```
        Write(Elements2);
    }
}
```

La méthode `Write` accepte une collection de type `IEnumerable<Classe1>` comme paramètre. La covariance permet d'appeler la méthode avec une collection de type `IEnumerable<Classe2>` car le type `Classe2` hérite du type `Classe1`.

b. La contravariance

La contravariance est la capacité pour une interface d'accepter des arguments génériques moins dérivés que ceux spécifiés. Pour illustrer la contravariance, l'exemple suivant utilise l'interface `IEqualityComparer<T>` qui est contravariante :

```
public class Classe1
{
    public string Prop1 { get; set; }
    public string Prop2 { get; set; }
}
public class Classe2 : Classe1 { }
public class Comparer : IEqualityComparer<Classe1>
{
    public int GetHashCode(Classe1 classe1)
    {
        return classe1.GetHashCode();
    }
    public bool Equals(Classe1 x, Classe1 y)
    {
        return x == y;
    }
}

static class Program
{
    [STAThread]
    static void Main()
    {
        IEqualityComparer<Classe1> classe1 = new Comparer();
        IEqualityComparer<Classe2> classe2 = classe1;
    }
}
```

La classe `Comparer` est utilisée pour effectuer des comparaisons entre des objets de type `Classe1`. Cette classe implémente l'interface `IEqualityComparer<T>` qui est contravariante et permet ainsi de réaliser une conversion implicite vers un type plus dérivé.

2. La création d'interfaces génériques variantes

Les interfaces génériques variantes sont déclarées à l'aide des mots clé `in` et `out` pour les paramètres de type. Pour déclarer un covariant de paramètre de type à l'aide du mot clé `out`, le type doit répondre aux conditions suivantes :

- Le type est utilisé seulement comme retour de méthode et non comme paramètre :

```
interface ITest<out T>
{
    T Get();                    // Autorisé
    void Set(T paramTest);      // Non autorisé
}
```

- Le paramètre de type ne doit pas être utilisé comme une contrainte pour les méthodes de l'interface. L'exemple suivant provoque une erreur de compilation :

```
interface ITest<out T>
{
```

```
    void MaMethode<U>() where U : T;
}
```

La création d'un contravariant de paramètre de type se fait à l'aide du mot clé `in`. Le type peut alors être utilisé comme paramètre de méthode ou dans une contrainte mais pas comme type de retour :

```
interface ITest<in T>
{
    T Get();                                // Non autorisé
    void Set(T paramTest);                  // Autorisé
    void MaMethode<U>() where U : T;        // Autorisé
}
```

La covariance et la contravariance peuvent être prises en charge par la même interface mais, dans ce cas, il s'agira de deux paramètres de type distinct :

```
interface ITest<out T1, in T2>
```

3. L'héritage d'interfaces génériques variantes

Il est bien entendu possible de créer une interface qui hérite d'une interface générique variante. Dans ce cas, il faut explicitement spécifier, avec les mots clés `in` et `out`, si l'interface dérivée prend en charge la variance. Le compilateur ne déduira pas la variance à partir de l'interface héritée :

```
interface IParent<out T> { }
interface IEnfant1<T> : IParent <T> { }
interface IEnfant2<out T> : IParent <T> { }
```

Dans l'interface `IEnfant1`, le paramètre de type est invariant car non explicitement spécifié alors que dans l'interface `IEnfant2`, le paramètre de type est covariant car le mot clé `out` est explicitement spécifié.

Il n'est pas autorisé de déclarer un paramètre de type covariant si la définition de l'interface de base précise qu'il est contravariant et inversement :

```
interface IParent<out T> { }
interface IEnfant<in T> : IParent <T> { }  // Non autorisé
```

La création de méthodes génériques

La déclaration des membres d'une classe générique se fait de la même manière que les membres de classes classiques avec en plus l'utilisation du paramètre `T` pour spécifier le type d'un paramètre qui est autorisé.

Ajoutez la méthode `Add` à la classe `ReportChangeList` :

```
public void Add(T Child)
{
    IReportChange child = (IReportChange)Child;
    child.Changed += new EventHandler(ChildChanged);
    this.children.Add(Child);
}
```

La méthode `Add` prend en paramètre un type répondant aux contraintes de la classe. L'objet passé en paramètre à la méthode implémente donc l'interface `IReportChange` et il peut être converti dans le type de l'interface ce qui permet à l'objet de type `ReportChangeList` de s'abonner à l'évènement `Changed` de l'objet passé en paramètre avant de l'ajouter à la liste `children`.

La liste va contenir de nombreux éléments et il faut donc implémenter une technique plus élaborée pour les différencier et assurer l'unicité de ceux-ci. Nous allons créer un indexeur basé sur une clé de type `string`. Cela implique dans un premier temps de créer une interface `IKey` dans le dossier **Library** :

```
public interface IKey
{
    string Key
    {
        get;
    }
}
```

Dans un second temps, ajoutez une contrainte sur la classe `ReportChangeList` de manière à obliger les types à implémenter l'interface `IKey` :

```
public class ReportChangeList<T> : IReportChildrenChange,
IEnumerable<T> where T : IReportChange, IKey
```

Ajoutez un indexeur à la classe `ReportChangeList` :

```
public T this[string Key]
{
    get
    {
        foreach (T aChild in this.children)
        {
            if (((IKey)aChild).Key.Equals(Key))
            {
                return aChild;
            }
        }
        return default(T);
    }
    set
    {
        for (int i = 0; i < this.children.Count; i++)
        {
            IKey aChild = (IKey)this.children[i];
            if (aChild.Key.Equals(Key))
            {
                this.children[i] = value;
                this.HasChanged = true;
                break;
            }
        }
    }
}
```

Nous avons créé un indexeur basé sur un paramètre de type `string`. Pour l'accesseur `get`, ce paramètre est comparé

avec tous les éléments de la liste afin de déterminer si l'un d'eux contient la même clé et ainsi le retourner. Si l'élément n'est pas trouvé, la valeur par défaut lui sera retournée. L'accesseur `set` fonctionne de la même manière mais lorsque l'élément a été trouvé, il est mis à jour.

Il n'y a plus qu'à modifier la méthode `Add` pour déterminer que la clé est unique parmi les éléments de la liste avant de faire l'ajout :

```
public void Add(T Child)
{
    IKey childKey = (IKey)Child;
    if (this[childKey.Key] == null)
    {
        IReportChange child = (IReportChange)Child;
        child.Changed += new EventHandler(ChildChanged);
        this.children.Add(Child);
    }
}
```

Afin de compléter la classe, il manque une méthode de suppression des éléments. La liste `children` étant protégée, les méthodes de suppression d'un élément de liste ne sont pas disponibles.

Créez une méthode `Remove` acceptant un paramètre de type `string` qui sera comparé aux clés des éléments de la liste afin de supprimer l'élément correspondant :

```
public void Remove(string Key)
{
    if (this[Key] != null)
    {
        this.children.Remove(this[Key]);
        this.HasChanged = true;
    }
}
```

La classe `ReportChangeList` se comporte désormais comme une liste de base avec nos propres fonctionnalités.

Valeur par défaut générique

En étudiant de près l'accesseur `get` de l'indexeur de la classe `ReportChangeList`, vous pouvez remarquer que si aucun élément de la liste ne correspond à la clé passée en paramètre, la valeur de retour est :

```
return default(T);
```

Un type générique peut concerner un type valeur ou un type référence, les types valeur n'étant pas `nullable` (ne pouvant pas avoir de valeur `null`) il est impossible de retourner `null` pour l'accesseur `get`. Le mot clé `default` est utilisé pour obtenir la valeur par défaut du paramètre type. Ainsi pour un paramètre de type référence, la valeur `null` sera retournée et pour un type valeur, il s'agira de sa valeur par défaut. Si `T` représente le type `int`, la valeur par défaut retournée sera zéro.

L’héritage de classe générique

Une classe générique peut hériter d’une autre classe. Les déclarations des classes enfants peuvent jouer avec les paramètres de type de la classe parente :

- La classe héritée peut garder le même paramètre de type :

```
class Parent<T>
{ }
class Enfant<T> : Parent<T>
{ }
```

- La classe héritée peut également spécifier le paramètre de type :

```
class Enfant : Parent<int>
{ }
```

- La classe héritée peut introduire de nouveaux paramètres de type :

```
class Enfant<T, T2> : Parent<T>
{ }
```

- Le nom du paramètre de type du parent peut être renommé et utilisé dans la déclaration du type enfant :

```
class Enfant<Tx, T2> : Parent<Tx>
{ }
```

Les délégués

Un délégué est une sorte de pont entre l'appelant d'une méthode et la méthode voulue. Les délégués se distinguent entre les types et les instances. Un type délégué définit le protocole auquel l'appelant et la méthode appelée doivent se conformer. Cela inclut la liste des paramètres et le type de retour, en un mot la signature. Une instance de délégué est un objet qui fait référence à une ou plusieurs méthodes qui ont une signature conforme.

La déclaration d'un délégué est précédée du mot clé `delegate` et la déclaration ne contient que la signature de la méthode comme pour un membre abstrait :

```
public delegate int Calcul(int i, int j);
```

Pour créer une instance de délégué, il suffit d'assigner une méthode dont la signature est conforme au délégué :

```
public class Classe1
{
    public Classe1(int i, int j)
    {
        Calcul C = new Calcul(Addition);
        int result = C.Invoke(i, j);
    }

    public int Addition(int i, int j)
    {
        return i + j;
    }
}
```

Dans l'exemple précédent, la méthode `Addition` pourrait avoir des surcharges. Le compilateur prendrait alors automatiquement la bonne surcharge en fonction de la signature du délégué auquel la méthode est assignée.

L'instruction d'instanciation du délégué dans l'exemple précédent peut être abrégée :

```
Calcul C = new Calcul(Addition);     // Notation complète
Calcul C = Addition;                 // Notation abrégée
```

De la même manière, l'invocation du délégué possède une notation abrégée :

```
C.Invoke(i, j);                      // Notation complète
C(i, j);                             // Notation abrégée
```

1. Les paramètres de méthodes

Les délégués peuvent être utilisés comme paramètres d'une méthode. L'implémentation la plus courante est de fournir une méthode de rappel qui pourra être exécutée à la fin de la méthode principale :

```
public delegate void Afficher(int i);

public class Classe1
{
    public void Addition(int i, int j, Afficher CB)
    {
        CB(i + j);
    }
    public void Affiche(int i)
    {
        Console.WriteLine(i);
    }

    public Classe1(int i, int j)
    {
        Afficher A = new Afficher(Affiche);
        Addition(i, j, A);
    }
}
```

2. Les méthodes cibles multiples

Comme indiqué précédemment, un délégué peut faire référence à une ou plusieurs méthodes. Les opérateurs += et -= permettent d'ajouter ou supprimer des références de méthode d'un délégué :

```
Calcul C = new Calcul(Addition);
C += Addition;      // Equivalent à C = C + Addition;
C -= Addition;      // Equivalent à C = C - Addition;
```

L'invocation du délégué lancera l'exécution de toutes les méthodes référencées dans l'ordre d'ajout.

Le délégué peut être initialisé avec la valeur `null`. L'ajout d'une référence de méthode est alors considéré comme une assignation d'une nouvelle valeur :

```
Calcul C = null;
C += Addition;          // Equivalent à C = Addition;
```

La suppression d'une référence de méthode avec l'opérateur -= sur un délégué qui ne contient qu'une seule méthode cible est équivalent à assigner la valeur `null` au délégué :

```
Calcul C = new Calcul(Addition);
C -= Addition;          // Equivalent à C = null;
```

Si un délégué possède un type de retour différent de `void`, l'appelant recevra la valeur de retour de la dernière méthode invoquée par le délégué. Les méthodes précédentes sont appelées mais leur valeur de retour est perdue.

Tous les délégués dérivent implicitement du type `System.MulticastDelegate` qui hérite du type `System.Delegate`. Le compilateur transforme les opérateurs +, -, += et -= effectués sur des délégués par les méthodes statiques `Combine` et `Remove` de la classe `System.Delegate`.

3. Les délégués génériques

Un délégué peut contenir des paramètres de type générique. Le type est alors précisé au moment de l'instanciation du délégué et la méthode cible doit se conformer à ce type :

```
public delegate T Calcul<T>(T i, T j);

public class Classe1
{
    public Classe1(int i, int j)
    {
        Calcul<int> C = new Calcul<int>(Addition);
        int result = C.Invoke(i, j);
    }

    public int Addition(int i, int j)
    {
        return i + j;
    }
}
```

4. La compatibilité des délégués

Les délégués ne sont pas compatibles entre eux même si leurs signatures sont identiques :

```
delegate void Delegue1();
delegate void Delegue2();

Delegue1 D1 = MaMethode;
Delegue2 D2 = D1;              // Non autorisé
```

Par contre, il est possible d'assigner un délégué à partir d'un autre qui possède la même signature avec la notation complète :

```
Delegue2 D2 = new Delegue2(D1);
```

Deux instances de délégués sont considérées comme égales si elles ont les mêmes méthodes cibles dans le même ordre :

```
Delegue1 d1 = MaMethode;
Delegue1 d2 = MaMethode;
// d1 == d2 vaut true
```

Lorsque vous appelez une méthode, vous pouvez fournir des arguments qui ont des types plus spécifiques que ceux spécifiés par la méthode. Il s'agit du polymorphisme. Pour les délégués, le processus est identique, il s'agit de la contravariance :

```
public delegate string Concat(string s1, string s2);

public class Classe1
{
    public Classe1(string s1, string s2)
    {
        Concat C = Concatene;
        string result = C(s1, s2);
    }
    public string Concatene(object o1, object o2)
    {
        return (string)o1 + (string)o2;
    }
}
```

Dans cet exemple, le délégué `Concat` est invoqué avec des paramètres de type `string`. Lorsque les arguments sont transmis à la méthode cible, ceux-ci sont simplement convertis dans le type `object`.

Lorsque vous appelez une méthode, vous pouvez renvoyer un type plus spécifique que celui demandé, il s'agit également de la mécanique du polymorphisme. Pour les délégués, le processus ne change pas, le type de retour du délégué peut être moins spécifique que le type de retour de sa méthode cible, il s'agit de la covariance :

```
public delegate object Concat(string s1, string s2);

public class Classe1
{
    public Classe1(string s1, string s2)
    {
        Concat C = Concatene;
        string result = (string)C(s1, s2);
    }
    public string Concatene(string s1, string s2)
    {
        return s1 + s2;
    }
}
```

Dans cet exemple, l'invocation du délégué retourne un type `object` malgré que la méthode cible renvoie un type `string`. Il faut alors convertir la valeur de retour pour obtenir le bon type.

Les évènements

En travaillant avec les délégués, deux concepts sont mis en évidence : la diffusion et la souscription. Le diffuseur est un type relié à un délégué. C'est lui qui décide quand le délégué doit être invoqué. Le souscripteur est l'ensemble des méthodes attachées à un délégué. Un souscripteur est totalement indépendant des autres souscripteurs même au sein d'un délégué. Les évènements formalisent ce schéma.

La manière la plus simple de déclarer un évènement est d'ajouter le mot clé `event` devant un membre délégué. L'évènement `Changed` dans l'interface `IReportChange` a été créé précédemment de cette manière :

```
event EventHandler Changed;
```

Le type qui contient le diffuseur a un accès total à celui-ci. Cela peut être l'ajout, la suppression ou l'exécution de méthodes cible. Les autres types ne pourront que souscrire à l'évènement avec les opérateurs `+=` et `-=`.

Analysez l'exemple suivant basé sur l'accesseur `set` de la propriété `HasChanged` de la classe `ProjectSettings` :

```
set
{
    if (this.hasChanged != value)
    {
        this.hasChanged = value;
        if (this.Changed != null)
            this.Changed(this, new EventArgs());
    }
}
```

La classe `ProjectSettings` déclenche l'évènement `Changed` à chaque fois que la valeur de la variable `hasChanged` est modifiée. Avant de déclencher l'évènement, il faut s'assurer qu'il n'est pas `null`, en d'autres termes, on s'assure que l'évènement possède des souscripteurs. Si vous essayez de déclencher un évènement qui n'a pas de souscripteur, une exception sera levée à l'exécution de l'application.

Si on enlevait le mot clé `event` de l'exemple, `Changed` deviendrait une variable de type délégué. Notre exemple fonctionnerait de la même manière mais le diffuseur serait moins sûr. En effet si vous utilisez un délégué à la place d'un évènement, les souscripteurs pourraient interférer les uns envers les autres de la manière suivante :

- En remplaçant les autres souscripteurs avec les opérateurs `+=` ou `-=`.
- Un souscripteur pourrait supprimer tous les autres en affectant `null` au délégué.
- Un souscripteur pourrait lancer la diffusion en invoquant directement le délégué.

Le Framework .NET définit un standard pour l'écriture des évènements. Le but est de fournir une consistance entre le Framework et le code utilisateur. Au cœur d'un évènement se trouve le type `System.EventArgs`. Ce type de base sert à transporter les informations entre le diffuseur et le souscripteur d'un évènement. Il est possible de dériver de ce type pour transmettre des informations aux souscripteurs.

Créez une nouvelle classe `ChangedEventArgs` dérivant de la classe de base `EventArgs` et exposant un membre `HasChanged` :

```
public class ChangedEventArgs : EventArgs
{
    public bool HasChanged { get; protected set; }
    public ChangedEventArgs(bool hasChanged)
    {
        this.HasChanged = hasChanged;
    }
}
```

Les types dérivant de `EventArgs` seront utilisés pour la transmission d'informations aux souscripteurs. Pour cette raison, les membres exposés sont soit en lecture seule, soit inaccessibles en écriture.

La prochaine étape de la création d'un évènement est de définir un délégué. Ce délégué doit répondre à trois règles :

- Le délégué ne doit pas avoir de type de retour (`void`).
- Le délégué doit accepter deux arguments : le premier sera de type `object` contenant l'objet qui aura déclenché

l'évènement et le second argument sera de type `EventArgs` ou tout autre type dérivé d'`EventArgs` contenant les informations supplémentaires.

- Le nom du délégué doit être suffixé par `EventHandler`.

Ces règles n'ont pour but que de garder une consistance avec les évènements du Framework .NET. Vous pourriez choisir un autre suffixe au délégué en remplacement de `EventHandler` tout comme les paramètres du délégué pourraient être différents.

Le Framework .NET définit un délégué générique `System.EventHandler<>` qui correspond à ces règles :

```
public delegate void EventHandler<TEventArgs>(object sender,
TEventArgs e) where TEventArgs : EventArgs;
```

Avant que les types génériques n'existent, les délégués devaient être créés comme ceci :

```
public delegate void ReportChangeEventHandler(object sender,
ChangedEventArgs e);
```

Pour des raisons historiques, de nombreux évènements du Framework .NET utilisent cette notation. Dans la suite des exemples, nous utiliserons la notation générique.

La dernière étape est la définition de l'évènement associé au type de délégué souhaité.

Modifiez la déclaration de l'évènement `Changed` dans l'interface `IReportChange` comme suit :

```
event EventHandler<ChangedEventArgs> Changed;
```

Cette modification entraîne des erreurs de compilation. D'une part parce que les types qui implémentent l'interface `IReportChange` ne possèdent plus la bonne signature pour l'évènement `Changed`. Il faut donc remplacer la signature des évènements dans les classes qui implémentent l'interface `IReportChange` par :

```
public event EventHandler<ChangedEventArgs> Changed;
```

D'autre part, la signature de la méthode `ChildChanged` de l'interface `IReportChildrenChange` doit être modifiée dans l'interface et dans les types qui l'implémentent :

```
void ChildChanged(object sender, ChangedEventArgs e);
```

Il faut également modifier la souscription des objets à l'évènement `Changed` afin de référencer le bon type de délégué :

```
Changed += new
           EventHandler<ChangedEventArgs>(ChildChanged);
```

Le dernier point à corriger est la modification du déclenchement des évènements qui doivent transmettre le bon type d'arguments :

```
this.Changed(this, new ChangedEventArgs(this.HasChanged));
```

Les expressions lambda

Une expression lambda est une méthode sans nom qui remplace une instance de délégué. Le compilateur transforme une expression lambda en un délégué.

La syntaxe d'une expression lambda est la suivante :

```
(paramètre1, paramètre2, ...) => expression ou instructions
```

Si l'expression ne contient qu'un seul paramètre, les parenthèses peuvent être omises comme dans l'exemple de la section suivante.

1. L'utilisation des expressions lambda

Prenons le délégué suivant :

```
public delegate int Multiplier(int i);
```

Il est possible d'assigner et d'invoquer une expression lambda comme ceci :

```
Multiplier M = x => x * 2;
int i = M(10);              // i = 20
```

Le compilateur résout une expression lambda de ce type en créant une méthode privée et en déplaçant l'expression dans cette méthode.

Chacun des paramètres de l'expression lambda correspond à un paramètre du délégué et le type de l'expression correspond au type de retour du délégué. Dans l'exemple précédent, `x` correspond au paramètre `i` du délégué et l'expression `x * 2` correspond au type de retour du délégué.

Une expression lambda peut contenir un bloc d'instructions à la place d'une expression :

```
public delegate int Absolue(int i);

Absolue A = x =>
{
    if (x < 0)
    {
        return -x;
    }
    else
    {
        return x;
    }
};
int i = A(10);                // i = 10
int j = A(-10);               // j = 10
```

En théorie, le compilateur déduit le type des paramètres d'une expression lambda suivant le contexte. Quand ce n'est pas le cas, vous devez spécifier explicitement le type de chacun des paramètres de la manière suivante :

```
Multiplier M = (int x) => x * 2;
```

2. Les délégués génériques

Les expressions lambda sont le plus communément utilisées avec les délégués génériques `Func` et `Action` :

```
Func<int, int> F = x => x * 2;
```

Les délégués génériques `Func` et `Action` sont fournis par le Framework .NET dans l'espace de noms `System`. Ces délégués sont extrêmement généraux, ils fonctionnent pour des méthodes qui retournent n'importe quel type d'objet et qui acceptent jusqu'à 16 arguments de type identique ou différent. Voici les signatures de quelques-uns :

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Et ainsi de suite jusqu'à 16 paramètres d'entrée

public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Et ainsi de suite jusqu'à 16 paramètres d'entrée
```

L'avantage principal des délégués génériques est de ne pas avoir à déclarer des délégués.

3. La capture de variable

Une expression lambda peut faire référence à des variables locales et des paramètres de la méthode dans laquelle elle est définie. Les variables externes qui sont référencées dans une expression lambda sont appelées des variables capturées :

```
public class Classe1
{
    public static int Test(int facteur)
    {
        Func<int, int> multiple = x => x * facteur;
        return multiple(10);
    }
}

int i= Classe1.Test(2);        // i = 20
```

Les variables capturées sont évaluées au moment où le délégué est invoqué et non pas au moment de la définition de l'expression lambda :

```
public class Classe1
{
    public static int Test(int facteur)
    {
        Func<int, int> multiple = x => x * facteur;
        facteur = 3;
        return multiple(10);
    }
}

int i= Classe1.Test(2);        // i = 30
```

Les variables capturées peuvent également être mises à jour par les expressions lambda :

```
int i = 0;
Action ajout = () => i++;
ajout();                       // i = 1
ajout();                       // i = 2
```

La portée d'une variable capturée est étendue à la portée du délégué. Dans l'exemple suivant, la variable `i` devrait être hors de portée et libérée à la fin de l'exécution de la méthode `Ajout`, mais comme la variable `i` et capturée, sa portée est allongée à la portée du délégué qui l'a capturée :

```
public class Classe1
{
    public static  Func<int> Ajout()
    {
        int i = 0;
        return () => i++;
    }
    public static void Test()
    {
        Func<int> ajout = Ajout();
        int i1 = ajout();            // i1 = 1
```

```
        int i2 = ajout();           // i2 = 2
    }
}
```

Une variable locale qui est instanciée dans une expression lambda est unique par invocation du délégué. Si on modifie l'exemple précédent pour obtenir la méthode suivante (on peut alors observer que le résultat est différent) :

```
public class Classe1
{
    public static Func<int> Ajout()
    {
        return () => { int i = 0; return i++; };
    }
    public static void Test()
    {
        Func<int> ajout = Ajout();
        int i1 = ajout();           // i1 = 0
        int i2 = ajout();           // i2 = 0
    }
}
```

Une variable définie dans une expression lambda possède la portée de l'invocation du délégué.

Lorsque la capture de variable concerne une variable d'itération dans un bloc `for` ou `foreach`, le compilateur traite ces variables comme déclarées en dehors de la boucle. Cela signifie que la même variable est capturée à chaque itération. Observez l'exemple suivant :

```
Func<string>[] F = new Func<string>[3];
for (int i = 0; i < 3; i++)
{
    F[i] = () => i.ToString();
}

string s = string.Empty;
foreach (Func<string> f in F)
{
    s += f();
}
```

Quelle est la valeur de la variable `s` ? La réponse est 333. L'explication est que dans la boucle `for`, il y a assignation dans le tableau `F` d'une expression lambda qui capture la variable d'itération `i`. Cette variable, qui devrait sortir de la portée au moment où la boucle `for` est terminée, reste en mémoire car elle est capturée. Lorsque la boucle `foreach` exécute les expressions lambda, `i` vaut 3 et à chaque itération la valeur 3 est ajoutée à la variable `s`.

La solution pour obtenir la vraie valeur de la variable d'itération serait de déclarer une variable dans la boucle `for` et de lui affecter la valeur de la variable d'itération. La variable capturée ne sera alors pas incrémentée et différente à chaque itération car nouvellement déclarée. L'exemple suivant permet d'obtenir la valeur 012 :

```
for (int i = 0; i < 3; i++)
{
    int j = i;
    F[i] = () => j.ToString();
}
```

Utiliser les formulaires

Les formulaires représentent l'interaction de l'utilisateur avec une application. Ils sont aussi bien utilisés pour la présentation que pour la saisie de données. La conception de l'interface utilisateur est une étape importante car une interface qui est visuellement cohérente et logique, donc ergonomique, est plus facile à prendre en mains.

1. Ajouter des formulaires au projet

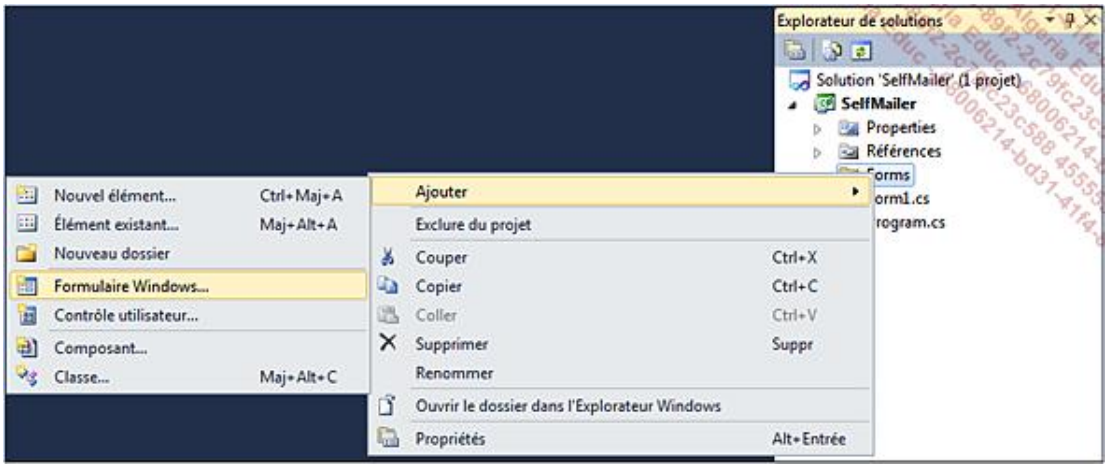
Lors de la création du projet d'application Windows, un formulaire par défaut a été créé, **Form1**. Ce formulaire représente la classe qui sera instanciée lors de l'exécution. Cette classe partielle est composée de deux fichiers **Form1.designer.cs** et **Form1.cs** :

- **Form1.designer.cs** : les fichiers de formulaires ayant l'extension **.designer.cs** sont générés automatiquement par le concepteur de vue. Lorsqu'un contrôle est déposé sur le formulaire, Visual Studio insère le code d'initialisation et les paramètres par défaut dans ce fichier. Le concepteur de vue est dans ce cas une façon de créer du code de manière visuelle.

- **Form1.cs** : ce fichier permet d'écrire la logique du formulaire, les traitements des actions utilisateurs et tout le code nécessaire au bon fonctionnement logique du formulaire.

Avant de créer un nouveau formulaire, nous allons créer un nouveau dossier nommé **Forms** à la racine du projet afin d'y stocker tous les formulaires. Séparer les éléments au sein d'un projet permet d'avoir une meilleure vue d'ensemble et de trouver plus facilement l'élément souhaité surtout en cas de maintenance.

En faisant un clic avec le bouton droit de la souris sur le nouveau dossier **Forms**, puis en sélectionnant **Ajouter** et **Formulaire Windows**, la boîte de dialogue d'ajout d'un élément de Visual Studio s'ouvre en ayant présélectionné le type **Windows Form**. Il suffit de saisir le nom du formulaire, **MailServerSettings.cs**, et de valider.



Le fichier **MailServerSettings.cs** créé par Visual Studio présente cette forme :

```
namespace SelfMailer.Forms
{
    public partial class MailServerSettings : Form
    {
        public MailServerSettings()
        {
            InitializeComponent();
        }
    }
}
```

On remarque que le nom du dossier contenant (**Forms**) a été ajouté à l'espace de noms du formulaire. En plus de classer les éléments visuellement, les dossiers permettent à Visual Studio d'organiser la hiérarchie des classes au sein d'espaces de noms lors de la création.

La classe partielle, identifiée par le mot clé `partial`, dérive de la classe `Form`, et permet au formulaire d'hériter de toutes les propriétés de base d'un formulaire Windows depuis le Framework .NET. Il est possible de faire hériter un formulaire d'un autre formulaire dans le cas où, par exemple, l'aspect reste le même mais le traitement serait différent.

La dernière remarque sur cette classe est que le constructeur comporte un appel à la méthode `InitializeComponent` qui se trouve dans le fichier **MailServerSettings.designer.cs** :

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "MailServerSettings";
}
```

Cette méthode se charge de lancer l'initialisation des éléments du formulaire et de définir les propriétés de ceux-ci. Il est déconseillé de modifier ce fichier avec l'éditeur de texte car les changements pourraient être écrasés par le designer lors de la génération de ce code.

2. Modifier le formulaire de démarrage

Le formulaire de démarrage correspond à celui qui est ouvert par défaut lorsque l'application est lancée. Il s'agit de la classe qui est instanciée dans le fichier **Program.cs**.

Pour modifier le formulaire de démarrage qui est actuellement **Form1**, il faut commencer par en créer un nouveau dans le dossier **Forms** et le nommer **Main.cs**. Dans le fichier **Program.cs**, il n'y a plus qu'à remplacer l'instruction :

```
Application.Run(new Form1());
```

par

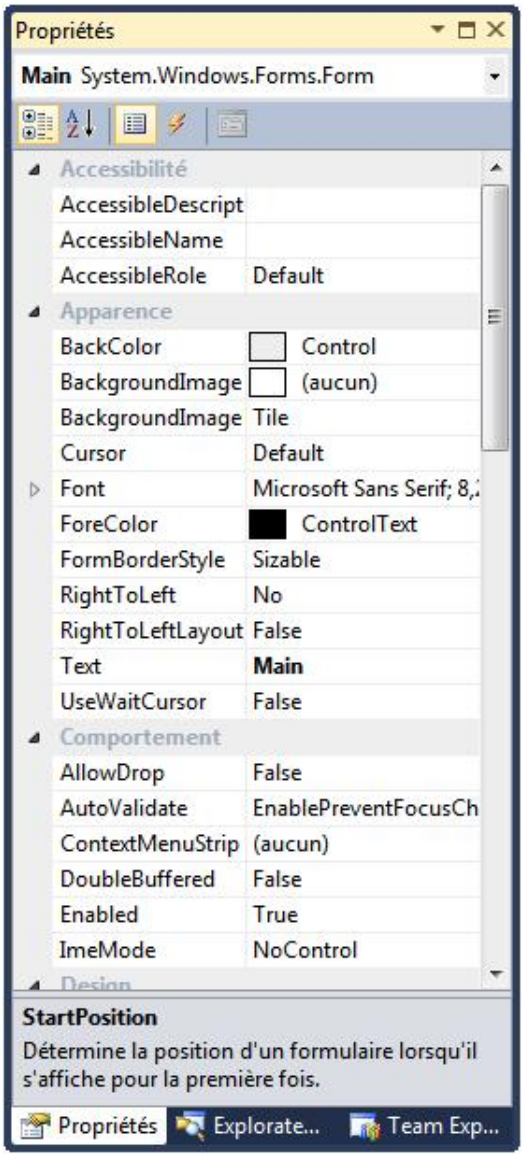
```
Application.Run(new Forms.Main());
```

Le formulaire **Form1** n'étant plus nécessaire, il peut être supprimé.

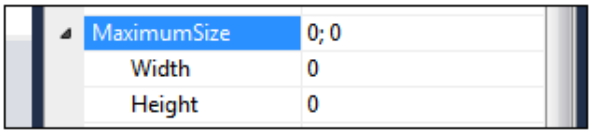
En lançant le débogage (**F5**), c'est bien le formulaire **Main** qui s'affiche.

3. Les propriétés des formulaires

En mode concepteur de vue, Visual Studio permet de définir les propriétés des objets grâce à la fenêtre **Propriétés** (**Ctrl + W, P**) :



Les propriétés sont classées par groupes logiques. Chacune d'elles affecte l'apparence ou le comportement du formulaire. Certaines propriétés, comme la propriété `MaximumSize`, sont des structures. En cliquant sur la flèche adjacente, il est possible de modifier séparément ces propriétés :



Les propriétés peuvent également être modifiées au niveau du code :

```
this.Text = "Paramètres du serveur mail";
```

Cette ligne placée dans le constructeur du formulaire aura le même effet que de modifier la propriété dans la fenêtre de propriétés du concepteur de vue à la différence que le Visual Studio aurait placé cette instruction dans le fichier designer.

L'affectation des propriétés suit la même syntaxe que pour d'autres membres de classe. L'opérateur d'assignation (=) est utilisé pour assigner une valeur à la propriété référencée par son nom.

Affectez les propriétés suivantes au formulaire **MailServerSettings** :

Propriété	Valeur	Effet
`FormBorderStyle`	FixedToolWindow	Le formulaire n'est plus redimensionnable. Les boutons d'agrandissement et de réduction ainsi

		que l'icône ne sont plus visibles.
ShowInTaskbar	False	Lorsque le formulaire sera ouvert, aucun onglet ne sera visible dans la barre des tâches Windows.
Size	345; 190	Fixe la taille du formulaire.
StartPosition	CenterParent	Le formulaire s'affichera de manière centrée par rapport à son parent.
Text	Paramètres du serveur mail	Le texte dans la barre de titre du formulaire est modifié.

Avec l'assignation des valeurs précédentes dans le concepteur de vue, Visual Studio a complété le fichier designer avec les lignes suivantes :

```
this.ClientSize = new System.Drawing.Size(339, 166);
this.FormBorderStyle =
            System.Windows.Forms.FormBorderStyle.FixedToolWindow;
this.ShowInTaskbar = false;
this.StartPosition =
            System.Windows.Forms.FormStartPosition.CenterParent;
this.Text = "Paramètres du serveur mail";
```

Tout ce qui est fait dans le concepteur de vue a une répercussion sur le code du fichier designer.

4. Les méthodes des formulaires

Les formulaires dérivant de la classe de base `System.Windows.Forms.Form` héritent de plusieurs méthodes permettant de gérer l'affichage des formulaires.

Une fois instancié, un formulaire a pour vocation d'être visible afin de permettre à l'utilisateur d'interagir avec celui-ci. Les méthodes `Show` et `ShowDialog` d'un formulaire permettent de le rendre visible. La méthode `Show` affiche le formulaire à l'écran et lui donne le focus. La propriété `Visible` du formulaire est alors automatiquement passée à `true`. La méthode `Show` ne crée pas le formulaire. Si celui-ci existe en mémoire mais n'est pas visible, c'est-à-dire que sa propriété `Visible` est égale à `false`, l'appel de la méthode revient à affecter la valeur `true` à la propriété `Visible`. La méthode `ShowDialog` est identique, à l'exception que le formulaire est affiché sous forme modale, c'est-à-dire qu'il doit être fermé avant qu'un autre formulaire ne puisse recevoir le focus de l'application. Cette méthode oblige donc l'utilisateur à effectuer une action.

Après que l'utilisateur ait effectué les actions voulues sur le formulaire, celui-ci doit être fermé. Il y a deux manières de fermer un formulaire. La première consiste à le masquer grâce à la méthode `Hide`. Cela équivaut à affecter la valeur `false` à la propriété `Visible` du formulaire. Avec cette méthode, le formulaire est toujours en mémoire et l'appel de la méthode `Show` le restitue à l'identique. La seconde manière de fermer un formulaire est la méthode `Close` qui supprime l'instance en mémoire et libère les ressources. Il devient impossible de faire appel à la méthode `Show` sur un formulaire après un appel à la méthode `Close` car le formulaire n'existe plus et il doit être à nouveau instancié. L'appel de la méthode `Close` sur le formulaire de démarrage entraîne la fermeture de l'application.

Ces méthodes seront utilisées lors de l'implémentation de gestionnaires d'évènements.

5. Les évènements des formulaires

Un formulaire, comme tout objet, a un cycle de vie et des moments clés déclenchant des évènements spécifiques.

L'évènement `Load` correspond à la création du formulaire en mémoire. Il est levé lorsque la méthode `Show`, ou `ShowDialog`, est appelée pour la première fois. Si le formulaire est déjà en mémoire, l'appel de la méthode `Show` ne déclenchera pas une seconde fois l'évènement `Load`. Un gestionnaire destiné à l'évènement `Load` permet d'initialiser les variables du formulaire à partir d'une base de données ou d'un fichier par exemple.

Les évènements `Activated` et `Deactivate` sont déclenchés lorsque le formulaire reçoit le focus et lorsqu'il le perd. Ce sont donc des évènements qui peuvent se déclencher plusieurs fois pendant la durée de vie du formulaire. Ces évènements ne se déclenchent que si le focus du formulaire est modifié au sein de l'application, cela signifie que si l'utilisateur donne le focus à une autre application puis revient, aucun des deux évènements ne sera levé.

Le processus de fermeture d'un formulaire entraîne le déclenchement de deux évènements distinct : `FormClosing` et `FormClosed`.

`FormClosing` est l'évènement qui est levé lorsque la fermeture du formulaire est déclenchée par l'appel à la méthode `Close` ou lorsque l'utilisateur clique sur le bouton de fermeture du formulaire. Un gestionnaire destiné à l'évènement `FormClosing` permet de vérifier et valider les données de l'utilisateur, le cas échéant, il est possible d'annuler la fermeture du formulaire et de le garder ouvert grâce à la propriété `Cancel` de l'objet `FormClosingEventArgs` passé en paramètre au gestionnaire d'évènement :

```
private void FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

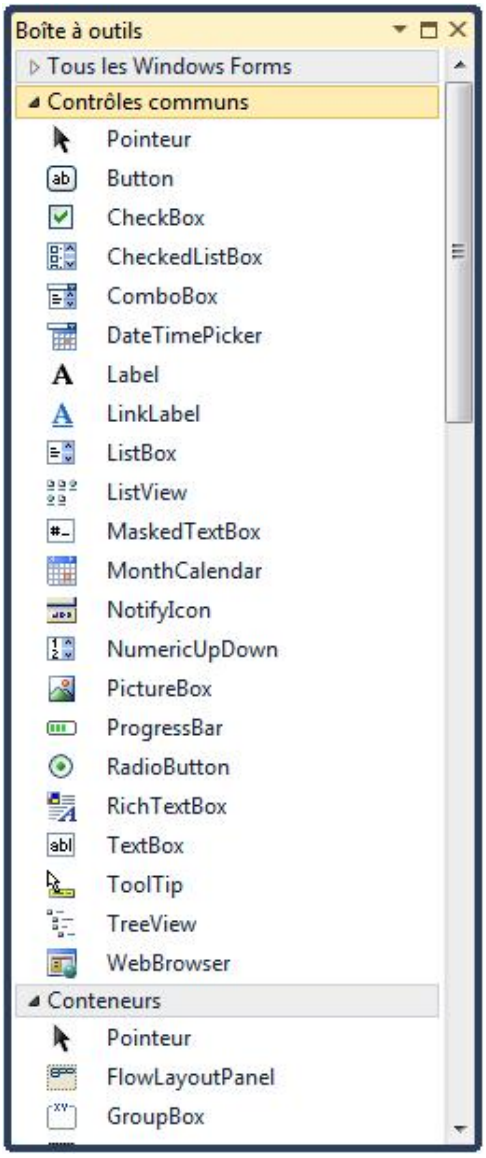
L'évènement `FormClosed` est déclenché après `FormClosing` et après l'exécution des gestionnaires d'évènements `FormClosing`, lorsque le formulaire est fermé.

Ces évènements seront utilisés lors de l'implémentation de gestionnaires d'évènements.

Utiliser les contrôles

1. Les types de contrôles

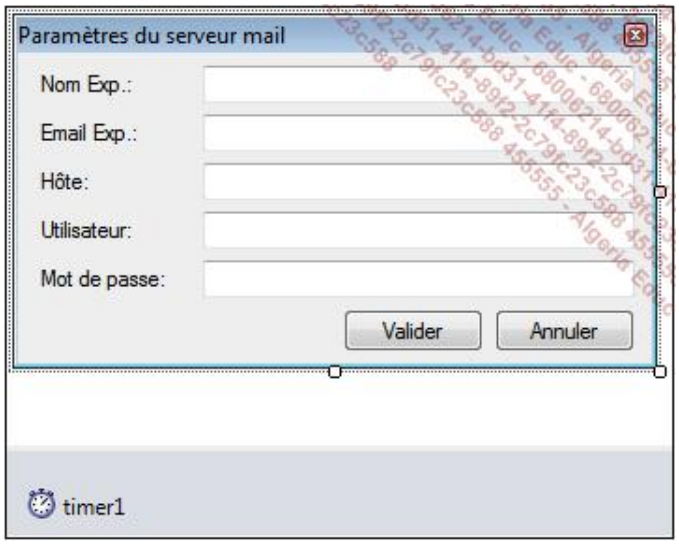
Le Framework .NET propose toute une série de contrôles et Visual Studio les intègre dans la **Boîte à outils** (**Ctrl** + **W**, **X**) du concepteur de vue :



La boîte à outils est subdivisée en groupes de contrôles : les contrôles communs, les conteneurs, les menus ou encore les données entre autres.

Chaque groupe comporte plusieurs types de contrôles : ceux qui vont essentiellement servir à l'utilisateur pour effectuer une action comme les **Button**, ceux qui vont permettre de saisir des données comme les **TextBox** et ceux qui vont donner des informations à l'utilisateur comme les **Label**.

Un autre type de contrôles, appelés des composants, ont la particularité de ne pas être visibles sur l'interface utilisateur à l'exemple du contrôle **Timer**. Les contrôles de ce type sont affichés dans la barre des composants du formulaire, en bas du concepteur de vue :

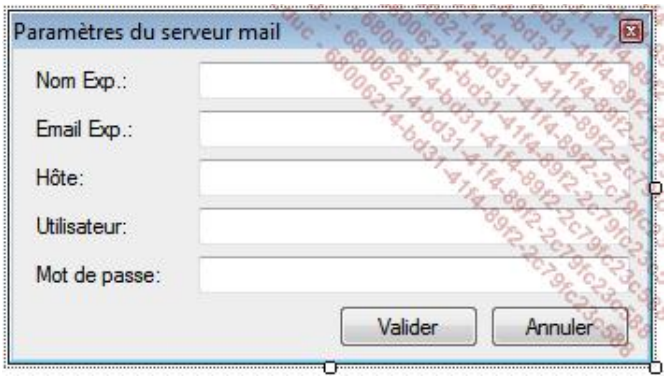


Les extendeurs sont un type de composant particulier qui ont pour but d'étendre les fonctionnalités des contrôles. Lorsqu'un extendeur est ajouté à un formulaire, tous les contrôles affichent une ou plusieurs nouvelles propriétés. Le contrôle **ErrorProvider** en est un exemple.

2. Ajouter des contrôles aux formulaires

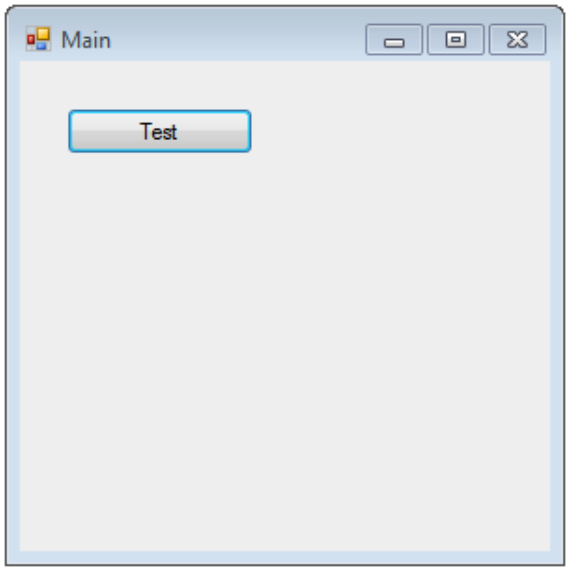
Pour ajouter un contrôle sur le formulaire, il suffit de le faire glisser depuis la boîte à outils et de le déposer sur le formulaire. Le concepteur de vue se charge d'écrire le code pour instancier le contrôle, le positionner et initialiser ses variables.

Ajoutez les contrôles au formulaire **MailServerSettings** afin d'obtenir ceci :



Il est également possible de créer un contrôle depuis le code en instanciant l'objet et en définissant ses propriétés puis en l'ajoutant à la collection de contrôles du formulaire. Le code suivant crée un contrôle de type **Button** et l'ajoute au formulaire courant :

```
Button newBtn = new Button();
newBtn.Name = "btnTest";
newBtn.Text = "Test";
newBtn.Location = new System.Drawing.Point(25, 25);
newBtn.Size = new System.Drawing.Size(100, 25);
this.Controls.Add(newBtn);
```



Tout contrôle conteneur, y compris les formulaires, possède une collection de contrôles enfants accessibles par la propriété `Controls`. Il est possible via cette collection d'ajouter des contrôles, d'en modifier ou de les supprimer dynamiquement pendant l'exécution comme dans l'exemple ci-dessus.

3. Les propriétés des contrôles

Tout comme les formulaires, les contrôles ont des propriétés qui permettent de les définir visuellement. Il suffit de sélectionner un contrôle sur le formulaire pour que la fenêtre des propriétés affiche celles de celui-ci.

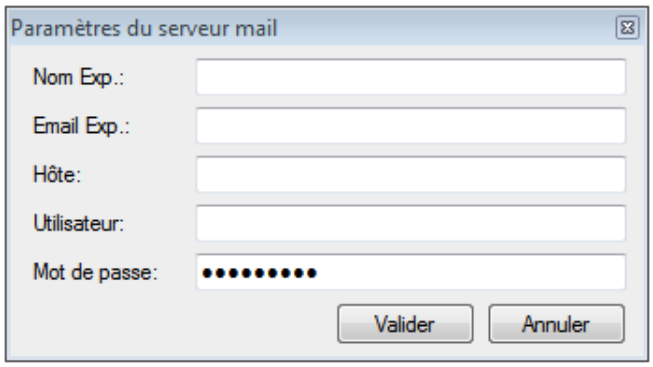
Spécifiez les propriétés des contrôles précédemment ajoutés au formulaire :

Type	Propriété	Valeur
`Label`	`Name`	lblFromName
	`Text`	Nom Exp.:
`Label`	`Name`	lblFromEmail
	`Text`	Email Exp.:
`Label`	`Name`	lblHost
	`Text`	Hôte:
`Label`	`Name`	lblUsername
	`Text`	Utilisateur:
`Label`	`Name`	lblPassword
	`Text`	Mot de passe:
`TextBox`	`Name`	FromName
`TextBox`	`Name`	FromEmail
`TextBox`	`Name`	Host
`TextBox`	`Name`	Username

TextBox	Name	Password
	UseSystemPasswordChar	True
Button	Name	Valid
	Text	Valider
Button	Name	Cancel
	Text	Annuler

Il est possible de modifier les propriétés de plusieurs contrôles en même temps en faisant glisser la souris autour de ceux-ci ou en maintenant la touche **Ctrl** et en cliquant sur les différents contrôles. La fenêtre propriété affiche seulement celles qui sont communes à tous les contrôles sélectionnés.

À noter la propriété `UseSystemPasswordChar` de la classe `TextBox`, transforme un champ de saisie sur une ligne classique en un champ de saisie pour mot de passe en utilisant les paramètres systèmes pour remplacer les caractères :



La propriété `PasswordChar` a le même effet dans le sens où le contrôle **TextBox** devient un champ de saisie de mot de passe mais au lieu d'utiliser le caractère par défaut du système sur lequel est exécutée l'application pour remplacer les vrais caractères, vous pouvez spécifier ce caractère. La propriété `UseSystemPasswordChar` est prioritaire sur `PasswordChar`, il faut donc que `UseSystemPasswordChar` soit à `false` pour que la propriété `PasswordChar` soit prise en compte.

4. Les menus

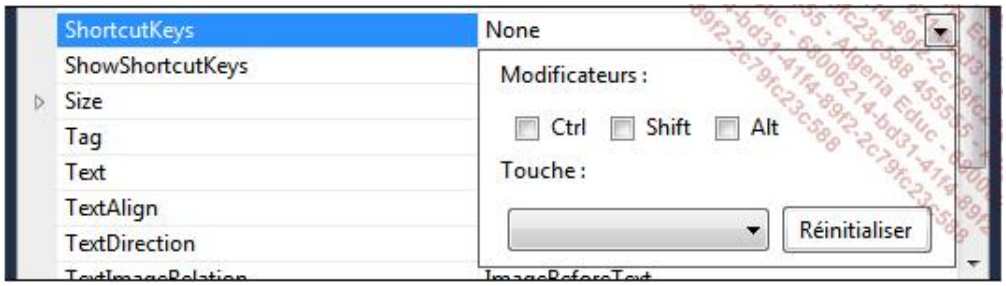
Toute application comportant plusieurs formulaires doit implémenter une interface utilisateur permettant de naviguer entre eux. Les menus représentent la manière la plus facile pour organiser et permettre à l'utilisateur d'accéder aux fonctionnalités de l'application.

La création d'un menu se fait avec le contrôle **MenuStrip**. Celui-ci contient une collection de contrôles **ToolStripMenuItem**. En ajoutant un contrôle **MenuStrip** au formulaire **Main**, il est par défaut ancré en haut du formulaire et prend toute la largeur de celui-ci. C'est la propriété Dock du contrôle qui permet d'obtenir ce comportement. Pour ajouter des éléments de menu dans le concepteur de vue, il suffit de cliquer sur le message **Tapez ici** et de saisir le texte du menu :

Le caractère **&** avant un caractère du titre du menu permet de spécifier la touche d'accès rapide que l'utilisateur pourra combiner avec la touche **Alt** pour accéder au menu. En ajoutant l'élément de menu ayant pour texte **&Paramètres** au formulaire **Main**, lors de l'exécution de l'application, l'utilisateur pourra ouvrir ce menu en tapant la combinaison de touches **Alt+P**. De manière visuelle, ce caractère est reconnaissable car il est souligné.

Les touches de raccourci permettent d'accéder directement aux menus sans passer par le menu parent comme pour la touche d'accès rapide. Il est possible d'assigner une combinaison de touches via la propriété `ShortcutKeys` de l'élément de menu. La fenêtre de propriété affiche une boîte intuitive permettant de définir la combinaison du

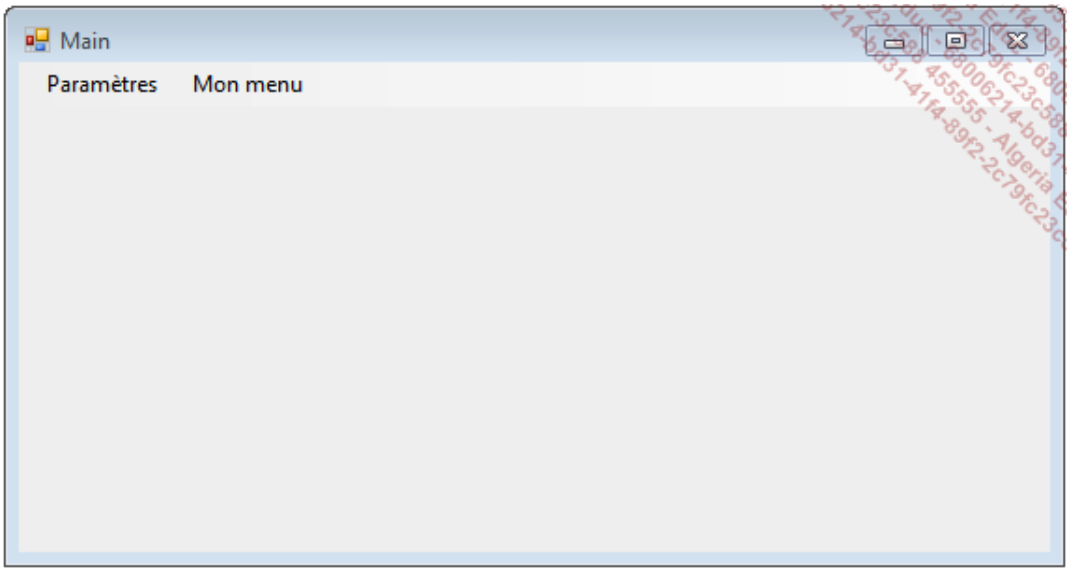
raccourci :



Ajoutez un élément de menu, nommé **Serveur &Mail** au menu **Paramètres** et attribuez-lui la combinaison **Ctrl+M** comme touche de raccourci. Le concepteur de vue affiche alors la combinaison à droite de l'élément de menu. Il est possible de faire disparaître ce texte en affectant la valeur `false` à la propriété `ShowShortcutKeys` ou de le modifier en insérant le texte à afficher dans la propriété `ShortcutKeyDisplayString` de l'élément de menu.

Il est également possible d'ajouter dynamiquement, lors de l'exécution, des menus pour apporter une interaction supplémentaire à l'utilisateur. Il suffit d'instancier un objet du type `ToolStripMenuItem` et de l'ajouter à la collection Items du menu parent :

```
ToolStripMenuItem newMenu = new ToolStripMenuItem("Mon menu");
this.MainMenu.Items.Add(newMenu);
```



Il existe un autre type de menus appelé menu contextuel (**ContextMenuStrip** dans la boîte à outils). Ces menus sont visibles par l'utilisateur lorsqu'il clique avec le bouton droit sur un élément de l'application. Les contrôles ont une propriété `ContextMenuStrip` permettant de définir quel est le menu contextuel à afficher. Mis à part cette différence, les menus contextuels fonctionnent de la même manière que les menus classiques que ce soit au niveau du concepteur de vue ou du code. Il est possible d'activer les touches de raccourcis mais pas les touches d'accès rapide :

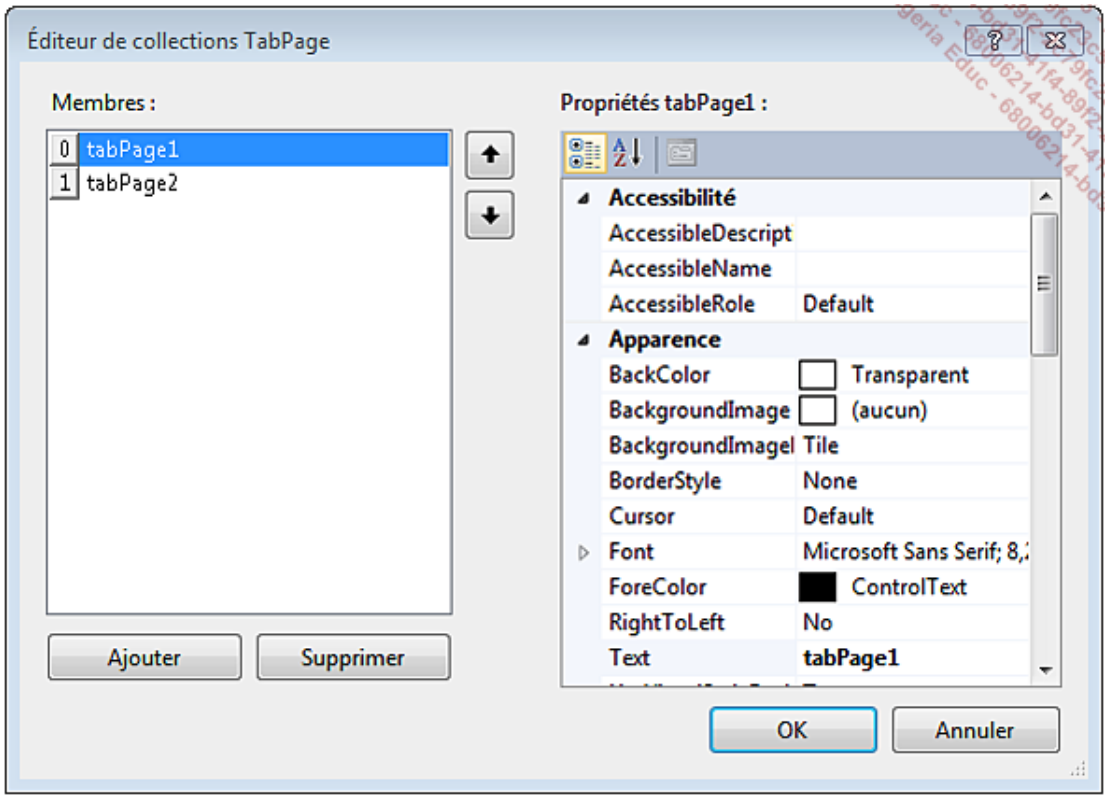
5. Les conteneurs

Certains contrôles, au même titre que les formulaires, ont pour vocation de regrouper d'autres contrôles. On les appelle des conteneurs. Le fait de modifier les propriétés d'un conteneur peut entraîner la modification des contrôles

qu'il contient. Par exemple, si la propriété `Enabled` d'un contrôle **Panel** est définie à `false`, tous les contrôles dans le **Panel** sont désactivés.

Le contrôle **TabControl** est un conteneur dont le rôle est de grouper d'autres contrôles au sein d'onglets dans la collection **TabPages**. Un **TabPage** est un conteneur ressemblant à un contrôle **Panel**. Il est possible d'activer les barres de défilement grâce à la propriété `AutoScroll`. Pour ajouter des contrôles à un **TabPage**, il suffit de sélectionner l'onglet puis de faire glisser les contrôles depuis la boîte à outils.

Ajoutez un contrôle **TabControl** sur le formulaire **Main** et définissez sa propriété `Dock` à la valeur `Fill`. Le contrôle s'étend pour prendre toute la place disponible sur le formulaire. En cliquant sur le bouton **...** de la propriété **TabPages**, le concepteur de vue affiche un formulaire permettant de gérer les objets **TabPage** :

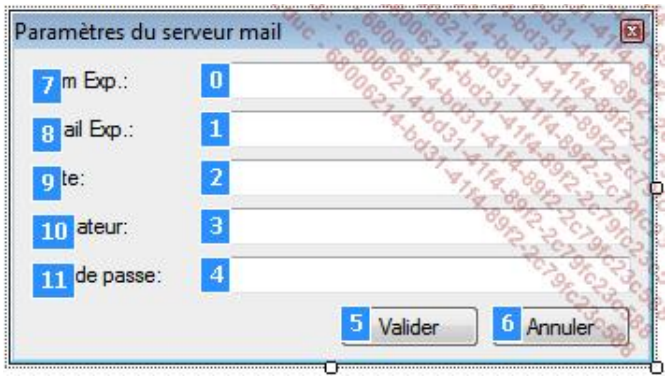


Pour l'application de démonstration, les **TabPage** seront ajoutés au contrôle de manière dynamique dans le code. Vous pouvez donc supprimer les deux **TabPage** par défaut.

6. L'ergonomie

L'ergonomie est un point très important d'un formulaire. L'information doit y être présentée de manière claire et intuitive. Visual Studio met à disposition des outils permettant d'améliorer l'expérience utilisateur.

L'ordre de tabulation représente la séquence des contrôles qui vont recevoir le focus en appuyant sur la touche **Tab**. Pour définir l'ordre, il faut définir la propriété `TabIndex` de chaque contrôle en partant de 0. Pour empêcher le contrôle de recevoir le focus avec la tabulation, il faut affecter la valeur `false` à la propriété `TabStop`. Plus le formulaire comportera de contrôles et plus il sera fastidieux de définir la propriété `TabIndex` de chacun d'eux. C'est pourquoi l'outil **Ordre de tabulation** dans le menu **Affichage** est très utile. En l'activant, les contrôles du formulaire affichent des numéros représentant la valeur de leur propriété `TabIndex` :



Pour définir un ordre différent que celui par défaut, il suffit de cliquer sur les contrôles les uns à la suite des autres afin d'obtenir une séquence différente. Par défaut la propriété `TabIndex` est définie dans l'ordre d'ajout des contrôles sur le formulaire.

Un formulaire comme **MailServerSettings**, qui est typiquement un formulaire de paramétrage avec un bouton de validation et un bouton d'annulation et qui sera ouvert avec la méthode `ShowDialog`, doit utiliser les propriétés `AcceptButton` et `CancelButton` du formulaire. Ces propriétés acceptent comme valeur le nom d'un bouton du formulaire et lors du clic sur ce bouton, le formulaire renvoie respectivement la valeur `DialogResult.OK` ou `DialogResult.Cancel` qui pourra être récupérée dans le formulaire parent pour effectuer des actions spécifiques :

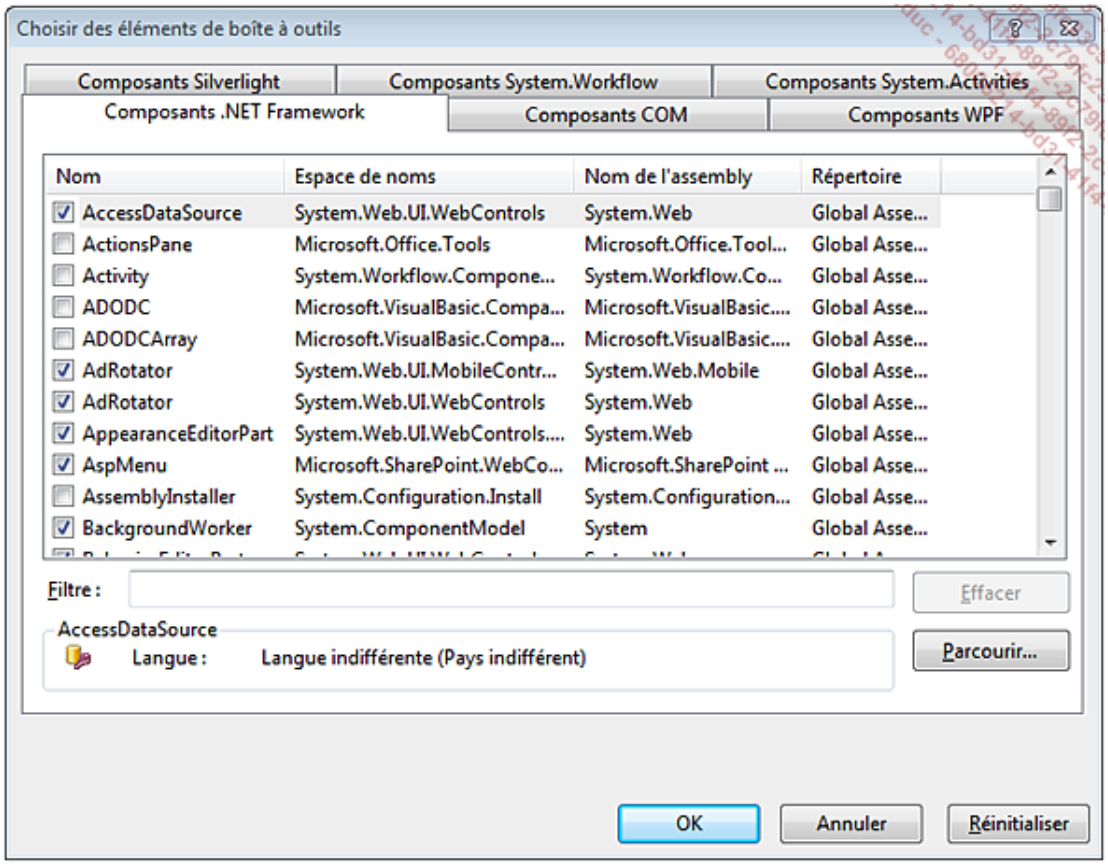
```
if (new MailServerSettings().ShowDialog() == DialogResult.OK)
{
}
```

7. Ajouter des contrôles à la boîte à outils

La boîte à outils n'est pas statique, il est possible d'ajouter de nouveaux contrôles provenant de développeurs tiers ou développés au sein d'une autre solution.

Pour ajouter un contrôle à la boîte à outils, un clic avec le bouton droit de la souris sur le groupe auquel on souhaite l'ajouter ouvre le menu contextuel.

Sélectionnez **Choisir les éléments...** et Visual Studio ouvre une fenêtre de sélection :



Si le contrôle n'est pas enregistré dans votre système, vous pouvez, en cliquant sur le bouton **Parcourir...**, sélectionner une librairie ou un exécutable qui contient le contrôle à ajouter. Il suffit ensuite de cocher les cases des contrôles à ajouter et de valider le formulaire.

Les contrôles créés dans un projet sont automatiquement ajoutés dans la boîte à outils sous un onglet nommé avec le nom du projet.

Introduction

Les évènements surviennent au cours de l'exécution de l'application. Chaque contrôle, y compris les formulaires, peut déclencher divers évènements suite à une action de l'utilisateur. Par exemple, lors du clic sur un contrôle de type `Button`, un évènement `Click` est déclenché. Si des méthodes, appelées gestionnaires d'évènements, sont abonnées à cet évènement, elles seront exécutées.

Chaque contrôle possède un évènement par défaut. Double cliquez sur un contrôle pour que Visual Studio crée le gestionnaire d'évènement associé, pour la plupart des contrôles. Il s'agit de l'évènement `Click`.

La création de gestionnaires d'évènements

Double cliquez sur l'élément de menu **Serveur Mail** du formulaire **Main**. Visual Studio ouvre le fichier de code associé avec l'implémentation de base du gestionnaire d'évènement :

```
private void MailServerMenu_Click(object sender, EventArgs e)
{
}
```

La méthode est marquée avec le mot clé `private`. Elle prend en premier paramètre un objet de type `object` qui représente l'objet qui a levé l'évènement et, en second paramètre, un objet de type `EventArgs` contenant des informations supplémentaires sur l'évènement.

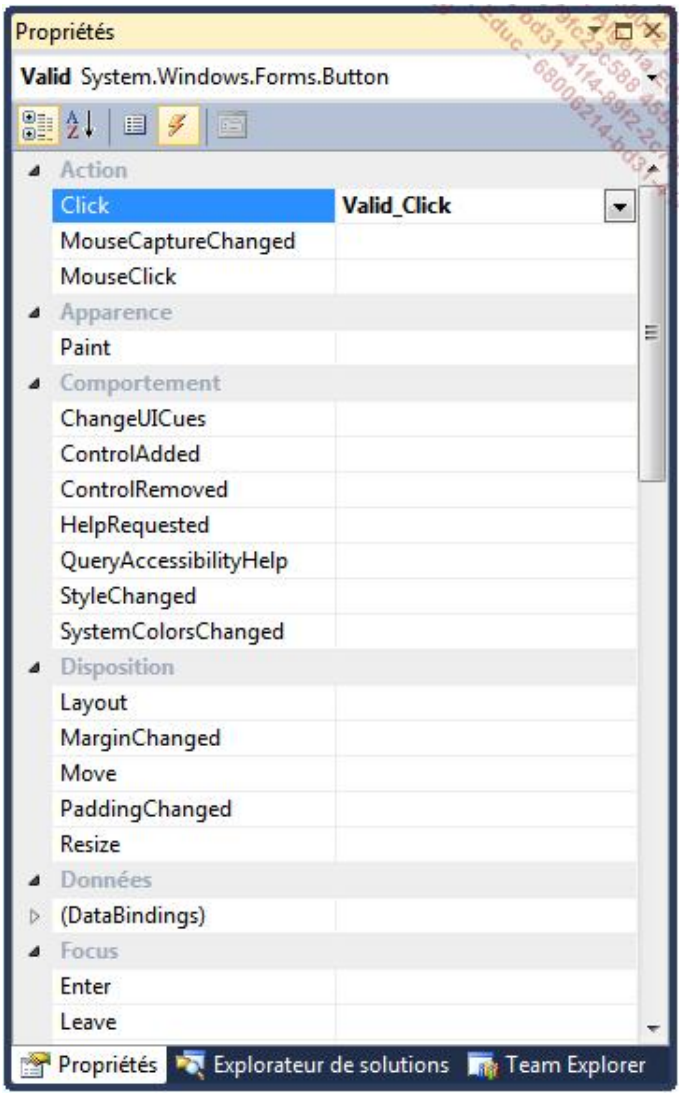
Ajoutez le code suivant au gestionnaire d'évènement :

```
new MailServerSettings().ShowDialog();
```

Cette instruction crée une nouvelle instance d'objet de type `MailServerSettings` et l'affiche de manière modale. Pour tester le gestionnaire d'évènement, il suffit de lancer l'application (**F5**) et de cliquer sur le menu **Serveur Mail**. Si vous utilisez le raccourci-clavier ou la touche d'accès rapide qui ont été définis, l'évènement est aussi déclenché.

Les contrôles ont de nombreux évènements, ils sont accessibles via la fenêtre de propriétés en cliquant sur l'icône **Évènements**  .

Il suffit alors de double cliquer sur l'évènement souhaité pour que Visual Studio génère le code associé. Créez un gestionnaire d'évènement pour le bouton **Valider** du formulaire **MailServerSettings** :



Comme précédemment, Visual Studio génère une méthode privée dans le fichier **MailServerSettings.cs** :

```
private void Valid_Click(object sender, EventArgs e)
{
}
```

1. La mécanique d'un évènement

Un gestionnaire d'évènements est une méthode invoquée via un délégué. Un délégué est un pointeur vers une méthode. Il permet de passer la référence du point d'entrée de cette méthode afin qu'elle puisse être invoquée de manière implicite.

Pour associer une méthode à un évènement, il faut créer une nouvelle instance du délégué pointant vers la méthode et l'associer à l'évènement.

Lors du déclenchement de l'évènement, les délégués abonnés seront invoqués ce qui équivaut à un appel de la méthode vers laquelle pointe le délégué.

La création des deux gestionnaires d'évènements précédents n'a pas seulement eu pour effet de créer des méthodes privées. Le concepteur de vue a également ajouté les instructions permettant d'abonner ces méthodes aux évènements dans la méthode `InitializeComponent` des fichiers **.designer.cs** des formulaires correspondants :

- L'instruction suivante a été ajoutée au fichier **Main.designer.cs** :

```
this.MailServerMenu.Click +=
          new System.EventHandler(this.MailServerMenu_Click);
```

- L'instruction suivante a été ajoutée au fichier **MailServerSettings.designer.cs** :

```
this.Valid.Click += new System.EventHandler(this.Valid_Click);
```

2. L'ajout dynamique d'un gestionnaire d'évènements

La déclaration des évènements peut se faire dynamiquement au moment de l'exécution. Lorsqu'un évènement est créé dans le concepteur de vue, Visual Studio insère dans le fichier designer du formulaire, au niveau de l'initialisation des contrôles, une ligne de code faisant l'association entre l'évènement et le délégué qui pointe vers la méthode qui aura le rôle de gestionnaire d'évènement.

L'opérateur `+=` permet de réaliser l'association d'un gestionnaire à un évènement. Cette méthode a été utilisée précédemment dans le constructeur de la classe `Project` :

```
public Project()
{
    this.ProjectSettings = new ProjectSettings();
    this.ProjectSettings.Changed +=
          new EventHandler<ChangedEventArgs>(ChildChanged);
    this.MailServerSettings = new MailServerSettings();
    this.MailServerSettings.Changed +=
          new EventHandler<ChangedEventArgs>(ChildChanged);
}
```

Ajouter un gestionnaire à un évènement est syntaxiquement identique à ajouter une méthode cible à un délégué.

3. La suppression dynamique d'un gestionnaire d'évènements

Tout comme l'ajout, il est possible de supprimer l'association d'une méthode à un évènement lors de l'exécution. Le code qui permet de supprimer cette association ressemble à celui qui sert à en ajouter, cette désassociation se fait avec l'opérateur `-=`. La référence à un délégué pointant vers la méthode à supprimer du gestionnaire suffit :

```
this.Valid.Click -= new EventHandler(this.Valid_Click);
```

Supprimer un gestionnaire à un évènement est syntaxiquement identique à supprimer une méthode cible à un délégué.

La suppression d’une association qui n’existerait pas n’entraîne pas la levée d’une erreur.

Les gestionnaires d'évènements avancés

1. Un gestionnaire pour plusieurs évènements

Un gestionnaire peut être abonné à plusieurs évènements différents. C'est utile lorsque plusieurs objets ou contrôles doivent déclencher la même action et cela évite de créer plusieurs gestionnaires identiques.

L'association se fait comme à l'habitude, en associant à l'évènement un délégué pointant vers la méthode, sauf que le délégué est identique pour les deux évènements :

```
this.Valid.Click += new EventHandler(this.Valid_Click);
this.Cancel.Click += new EventHandler(this.Valid_Click);
```

Vous pouvez aussi commencer par déclarer et instancier le délégué puis l'associer à plusieurs évènements :

```
EventHandler newHandler = new EventHandler(this.Valid_Click);
this.Valid.Click += newHandler;
this.Cancel.Click += newHandler;
```

Comme plusieurs contrôles déclenchent le même évènement, la distinction des actions à effectuer peut se faire grâce au paramètre `sender` qui est passé au gestionnaire d'évènement :

```
private void Valid_Click(object sender, EventArgs e)
{
    MessageBox.Show((sender as Control).Name);
}
```

2. Plusieurs gestionnaires pour un évènement

À l'inverse, il est possible d'associer plusieurs gestionnaires pour un même évènement. L'association se fait de manière habituelle avec l'opérateur `+=` :

```
this.Valid.Click += new System.EventHandler(this.Valid_Click1);
this.Valid.Click += new System.EventHandler(this.Valid_Click1);
this.Valid.Click += new System.EventHandler(this.Valid_Click2);
```

L'ordre d'appel des gestionnaires quand l'évènement est déclenché est le même que l'ordre d'association. Ainsi lorsqu'un évènement est associé avec les gestionnaires A, B et C dans cet ordre, le déclenchement de l'évènement impliquera l'appel des gestionnaires dans ce même ordre A puis B et enfin C.

Dans le code ci-dessus, il y a association de trois gestionnaires pour le même évènement. Les deux premiers délégués sont identiques. Ils pointent vers la méthode `Valid_Click1`. Lorsque l'évènement `Click` sera déclenché, la méthode `Valid_Click1` sera appelée successivement deux fois puis ce sera la méthode `Valid_Click2`. Si la méthode `Valid_Click2` supprime l'association entre l'évènement et le délégué qui est en doublon comme suit :

```
private void Valid_Click2(object sender, EventArgs e)
{
  this.Valid.Click -= new System.EventHandler(this.Valid_Click1);
}
```

une seule des deux associations entre l'évènement `Click` et la méthode `Valid_Click1` sera supprimée, la première. Il en restera donc une et l'évènement sera désormais associé à deux gestionnaires au lieu de trois.

Introduction

Le principe général de toute application est de prendre des données en entrée et de fournir des données traitées en sortie. Ces données peuvent provenir de différentes sources et avant de pouvoir les traiter, il faut les valider afin de s'assurer de leur format et ainsi améliorer la fiabilité de l'application.

Dans le cas des formulaires et donc de données saisies par un utilisateur, la validation et le contrôle vont pouvoir se faire au niveau des champs de saisie et aussi au niveau du formulaire. En plus de la validation et donc de l'interception des erreurs de saisie, il faut faire un retour explicatif pour l'utilisateur afin que celui-ci puisse corriger ses données.

La validation au niveau des champs

1. Les propriétés de validation

La validation au niveau des champs vérifie les données saisies au fur et à mesure par l'utilisateur. Certaines propriétés des champs vont permettre de limiter la saisie de l'utilisateur à l'exemple de la propriété `MaxLength` des contrôles de type `TextBox`. Cette propriété, une fois définie empêche l'utilisateur de saisir plus de caractères. On a donc une première validation native des contrôles de type `TextBox` qui évite d'écrire du code supplémentaire pour détecter le nombre de caractères saisis.

Les propriétés des contrôles permettent de faire de simples limitations dans la saisie sans pour autant gérer tous les cas. Si un champ doit avoir comme entrée un code postal, la propriété `MaxLength` du champ de type `TextBox` pour la saisie de cette donnée permettra de limiter l'utilisateur à la saisie de 5 caractères au maximum mais rien ne l'empêchera d'en saisir moins ou de saisir un autre caractère que des chiffres. Les évènements liés au clavier seront donc utiles pour valider la saisie.

2. Les évènements de validation

Les évènements claviers comme, `KeyDown`, `KeyUp` ou `KeyPress` sont déclenchés pour le contrôle qui a le focus au moment de la saisie et suivant le cas lorsqu'une touche est enfoncée, relâchée ou pressée, c'est-à-dire enfoncée puis relâchée. Lorsque ces évènements sont déclenchés, l'objet de type `KeyEventArgs`, ou `KeyPressEventArgs` pour l'évènement `KeyPress`, passé en paramètre au gestionnaire de l'évènement, permet d'obtenir des informations sur la touche qui a déclenché l'évènement.

a. KeyDown et KeyUp

Les évènements `KeyDown` et `KeyUp` servent principalement à déterminer si les touches **Alt**, **Ctrl**, **Suppr** ou **Shift** ont été enfoncées ou relâchées. L'objet `KeyEventArgs` passé au gestionnaire expose des propriétés booléennes pour déterminer quelle est la combinaison de touches qui a déclenché l'évènement. La propriété `KeyCode` permet de déterminer le code de la touche qui a déclenché l'évènement et ainsi effectué une action en fonction de celui-ci.

b. KeyPress

L'évènement `KeyPress` est déclenché par les touches correspondantes aux caractères alphanumériques ainsi que certains caractères spéciaux tels que les touches **Enter** ou **Del**. Le principe est que si une touche génère une valeur ASCII, elle déclenche l'évènement. C'est pourquoi les touches comme **Ctrl** ou **Alt** ne déclenchent pas cet évènement contrairement aux évènements `KeyDown` et `KeyUp`.

La propriété `KeyChar` de l'objet `KeyPressEventArgs` passé au gestionnaire de l'évènement `KeyPress` permet de déterminer le caractère qui a été saisi. Pour déterminer si la touche pressée est bien un chiffre, il suffit de tester ce caractère dans le gestionnaire d'évènement lié.

Le type `Char` contient les méthodes statiques permettant de déterminer le type de caractère passé en paramètre :

```
private void TextBox_KeyPress(object sender, KeyPressEventArgs e)
{
    if (!Char.IsDigit(e.KeyChar))
        MessageBox.Show("Le caractère pressé n'est pas un chiffre");
}
```

c. Validating et Validated

Certaines données doivent pouvoir être vérifiées dans leur ensemble et pas seulement caractère par caractère car elles n'ont du sens qu'une fois complètes. Les évènements `Validating` et `Validated` sont des évènements levés lorsque le contrôle perd le focus et donc que la saisie est terminée pour celui-ci.

Chaque contrôle capable de recevoir une entrée utilisateur, que ce soit au clavier ou à la souris, peut recevoir le focus. Le contrôle qui a le focus est celui qui reçoit les entrées utilisateur. Il ne peut y en avoir qu'un à la fois dans une application. Une fois que le contrôle perd le focus et avant de le passer au contrôle suivant, l'évènement `Validating` est levé. Cet évènement est levé uniquement si le contrôle qui est sur le point de recevoir le focus a la propriété `CausesValidation` définie à `true` (valeur par défaut). L'évènement `Validating` permet de faire la validation

du contrôle une fois que l'utilisateur a fini sa saisie. Dans le cas où la donnée du contrôle ne serait pas conforme, le gestionnaire de l'évènement peut utiliser la propriété `Cancel` de l'objet `CancelEventArgs` passé en paramètre pour annuler la perte de focus du contrôle :

```
private void TextBox_Validating(object sender, CancelEventArgs e)
{
    e.Cancel = true;
}
```

Cette technique est à utiliser avec prudence puisque le focus restera sur le contrôle tant que la saisie ne sera pas correcte. Ainsi, le test permettant de définir si la saisie est valide ou non doit être bien conçu de manière à ne pas bloquer indéfiniment le focus sur ce contrôle.

Dans le cas où la validation effectuée avec le gestionnaire de l'évènement `Validating` est bonne, l'évènement `Validated` est déclenché permettant d'effectuer toute action avec les données validées.

Dans le formulaire **MailServerSettings**, le contrôle de type `TextBox` pour l'adresse email de l'expéditeur doit correspondre à un format précis.

Ajoutez un gestionnaire pour l'évènement `Validating` de ce contrôle et faites un test pour déterminer si l'adresse email est syntaxiquement correcte :

```
void FromEmail_Validating(object sender, CancelEventArgs e)
{
    string pattern = @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\." +
                     @"[0-9]{1,3}\.[0-9]{1,3}\.)|" +
                     @"(([a-zA-Z0-9\-]+\.)+))" +
                     @"([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
    Regex reg = new Regex(pattern);
    if (!reg.IsMatch(this.FromEmail.Text))
    {
        MessageBox.Show("Le format de l'email est incorrect.");
        e.Cancel = true;
    }
 }
```

Cette méthode utilise les expressions régulières (dans l'espace de noms `System.Text.RegularExpressions`), qui seront détaillées plus loin dans l'ouvrage, pour déterminer si le format de la saisie correspond à une adresse email.

La validation au niveau du formulaire

La validation au niveau du formulaire consiste à effectuer le test des données saisies en une seule fois avant la fermeture du formulaire. Le principe est de parcourir tous les contrôles afin de déterminer quelles sont les erreurs et de les afficher à l'utilisateur en une seule fois. De plus, certaines données d'un formulaire n'ont de sens que si elles sont mises ensemble et pas seulement les unes séparées des autres. En outre, un champ qui ne doit pas rester vide, et qui n'aurait pas reçu le focus, pourrait être non rempli au moment de la fermeture du formulaire.

Un formulaire, possédant un bouton de validation, permet d'effectuer les contrôles souhaités pour valider les données dans un gestionnaire répondant à l'évènement `Click` du bouton de validation. Le formulaire **MailServerSettings** contient le champ permettant de spécifier l'hôte du serveur mail qui ne peut pas être vide.

Ajoutez le code de validation de ce champ au gestionnaire de l'évènement `Click` du bouton **Valid** créé précédemment :

```
void Valid_Click(object sender, EventArgs e)
{
    if (string.IsNullOrWhiteSpace(this.Host.Text))
        MessageBox.Show("Le champ Hôte doit être complété.");
}
```

Ce code ne valide pas les champs un par un. Si le focus ne parvient jamais au contrôle de saisie de l'email expéditeur, sa méthode de validation, `FromEmail_Validating`, ne sera jamais exécutée. Pour lancer la validation des contrôles enfants du formulaire, ajoutez le code suivant à la méthode `Valid_Click` :

```
else if (this.ValidateChildren())
{
    // Code d'enregistrement des données
    this.DialogResult = DialogResult.OK;
}
```

L'appel de la méthode `ValidateChildren` du formulaire lève l'évènement `Validating` de tous les contrôles enfants du formulaire et renvoie `true` si aucun des contrôles ne contient d'erreur de saisie. Dans le cas d'une saisie correcte, il y aura enregistrement des données saisies puis fermeture du formulaire en affectant la valeur `DialogResult.OK` à la propriété `DialogResult` du formulaire. Cette affectation indique au formulaire de se fermer et ce sera la valeur retournée par la méthode `ShowDialog` qui aura ouvert le formulaire.

Avant de fermer le formulaire, les données doivent être conservées de manière à ce qu'à la prochaine ouverture du formulaire, les champs soient pré remplis.

Ajoutez un membre statique de type `Project` à la classe `Program` :

```
public static Library.Project Project;
```

Ce champ est accessible depuis tout le projet et c'est cet objet qui contiendra les modifications de l'utilisateur. La première chose à faire est de l'initialiser au sein de la méthode `Main` de la classe `Program` :

```
Project = new Library.Project();
```

Pour enregistrer les valeurs du formulaire, ajoutez les membres suivants à la classe `MailServerSettings` :

```
protected string fromName;
protected string fromEmail;
protected string host;
protected string username;
protected string password;

public string FromName
{
    get { return this.fromName; }
    set
    {
        if (this.fromName != value)
        {
            this.fromName = value;
            this.HasChanged = true;
        }
    }
}
public string FromEmail
```

```
{
    get { return this.fromEmail; }
    set
    {
        if (this.fromEmail != value)
        {
            this.fromEmail = value;
            this.HasChanged = true;
        }
    }
}
public string Host
{
    get { return this.host; }
    set
    {
        if (this.host != value)
        {
            this.host = value;
            this.HasChanged = true;
        }
    }
}
public string Username
{
    get { return this.username; }
    set
    {
        if (this.username != value)
        {
            this.username = value;
            this.HasChanged = true;
        }
    }
}
public string Password
{
    get { return this.password; }
    set
    {
        if (this.password != value)
        {
            this.password = value;
            this.HasChanged = true;
        }
    }
}
```

Il est maintenant possible de sauver les valeurs saisies dans l'objet `Project` :

```
Program.Project.MailServerSettings.FromName = FromName.Text;
Program.Project.MailServerSettings.FromEmail = FromEmail.Text;
Program.Project.MailServerSettings.Host = Host.Text;
Program.Project.MailServerSettings.Username = Username.Text;
Program.Project.MailServerSettings.Password = Password.Text;
```

La dernière action pour compléter la mécanique du formulaire **MailServerSettings** est de restaurer les données lors de l'ouverture du formulaire. Ajoutez le code suivant dans le constructeur du formulaire :

```
FromName.Text = Program.Project.MailServerSettings.FromName;
FromEmail.Text = Program.Project.MailServerSettings.FromEmail;
Host.Text = Program.Project.MailServerSettings.Host;
Username.Text = Program.Project.MailServerSettings.Username;
Password.Text = Program.Project.MailServerSettings.Password;
```

Vous pouvez tester le formulaire en lançant l'application (**F5**).

Les méthodes de retour à l'utilisateur

Lorsqu'une donnée est incorrecte, il faut informer l'utilisateur de son erreur afin qu'il puisse la corriger. Si l'erreur est assez explicite, attirer l'attention de l'utilisateur sur un contrôle du formulaire en changeant sa couleur en rouge peut suffire. Dans les exemples précédents, la méthode `MessageBox.Show` a été utilisée pour afficher un message à l'utilisateur. Cela permet de décrire l'erreur et d'être plus précis. Le composant `ErrorProvider` quant à lui permet de signaler les erreurs d'une manière plus élégante.

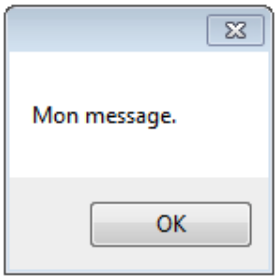
1. MessageBox

Le type `MessageBox` contenant la méthode statique `Show` est la manière la plus simple d'afficher un message à l'utilisateur. Cette méthode affiche une boîte modale ce qui a pour effet de stopper l'exécution de l'application et l'utilisateur est contraint d'effectuer une action pour la fermer.

La manière la plus simple d'afficher un message est d'utiliser la méthode `Show` avec un paramètre unique de type `string` :

```
MessageBox.Show("Mon message.");
```

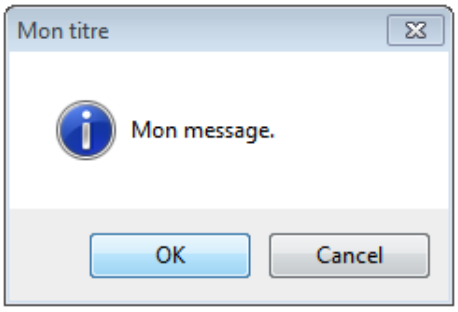
La boîte de dialogue affichée sera :



La méthode `Show` comporte 20 surcharges permettant de personnaliser le message affiché. Il est ainsi possible de modifier le titre, d'afficher une icône en fonction du type de message transmis, de choisir les boutons qui seront affichés et de définir lequel sera le bouton par défaut parmi les paramètres possibles :

```
MessageBox.Show("Mon message.",
                "Mon titre",
                MessageBoxButtons.OKCancel,
                MessageBoxIcon.Information,
                MessageBoxDefaultButton.Button1);
```

La boîte de dialogue affichée sera :



2. ErrorProvider

Le composant `ErrorProvider` est une manière plus élégante que le type `MessageBox` pour afficher les messages à l'utilisateur. Il permet de définir un message d'erreur en cas de saisie non valide pour chacun des contrôles du formulaire. Le message d'erreur se présente sous la forme d'une icône placée à côté du contrôle et affiche une info bulle au passage de la souris afin de détailler l'erreur.

Faites glisser depuis la boîte à outils (dans le groupe **Composants**) un contrôle **ErrorProvider** et déposez-le sur le

formulaire **MailServerSettings**. Le composant apparaît dans la zone des composants en bas du concepteur de vue. Renommez-le en `errorProvider`. Désormais, tous les contrôles du formulaire ont de nouvelles propriétés car le composant **ErrorProvider** est un extendeur, c'est-à-dire qu'il étend le champ d'action des contrôles en leur attribuant de nouvelles propriétés et/ou méthodes en fonction du composant.

Le composant **ErrorProvider** ne contient pas beaucoup de propriétés et elles sont rapidement paramétrables via le concepteur de vue. La propriété `Icon` permet de spécifier l'image qui sera affichée à côté du contrôle qui a une erreur. Les propriétés `BlinkRate` et `BlinkStyle` permettent de définir de quelle manière l'image va clignoter et à quelle vitesse.

Définissez la propriété `BlinkStyle` du composant `errorProvider` avec la valeur `NeverBlink` pour que l'icône s'affichant en cas d'erreur ne clignote pas.

Le composant **ErrorProvider** a ajouté 3 nouvelles propriétés aux contrôles qui sont reconnaissables car elles comportent le nom de l'extendeur :

- `Error sur errorProvider` : cette propriété contient la description de l'erreur pour le contrôle. Si elle est définie, l'icône d'erreur s'affiche.
- `IconAlignment sur errorProvider` : cette propriété permet de définir la position de l'icône d'erreur par rapport au contrôle.
- `IconPadding sur errorProvider` : la valeur de cette propriété définit l'espace entre le contrôle et l'icône d'erreur.

L'affectation d'une erreur à un contrôle au moment de l'exécution se fait d'une manière spécifique au composant **ErrorProvider**. Il faut utiliser sa méthode `SetError` en passant en paramètres le nom du contrôle et le texte descriptif de l'erreur.

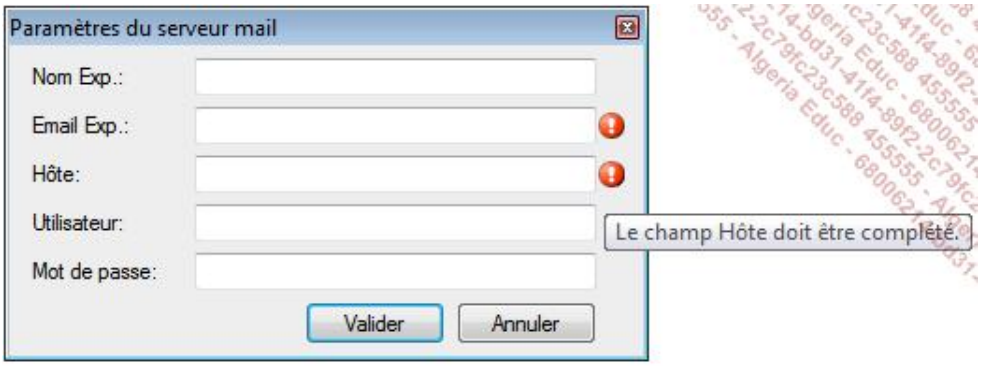
Dans la méthode `FromEmail_Validating` du formulaire **MailServerSettings**, modifiez la méthode d'affichage du message d'erreur par :

```
this.errorProvider.SetError(this.FromEmail,
                        "Le format de l'email est incorrect.");
```

Faites de même pour l'affichage du message d'erreur dans la méthode `Valid_Click` :

```
this.errorProvider.SetError(this.Host,
                        "Le champ Hôte doit être complété.");
```

Lancez l'exécution de l'application (**F5**) pour observer le résultat :



Introduction

Le développement d'applications est principalement basé sur les contrôles, ils fournissent des fonctionnalités distinctes sous une forme visuelle permettant à l'utilisateur d'interagir avec eux. Tous ces contrôles dérivent à un niveau plus ou moins lointain de la classe de base `System.Windows.Forms.Control`. Visual Studio propose l'intégration de contrôles tiers par l'ajout à la boîte à outils. Mais si le besoin est très spécifique, il est possible de créer ses propres contrôles.

La classe de base des contrôles, `Control`, fournit les fonctionnalités de base qui sont nécessaires, notamment pour les entrées utilisateurs via le clavier et la souris. Cela implique donc des propriétés, des méthodes et des évènements communs à tous les contrôles. Néanmoins, cette classe de base ne fournit pas la logique d'affichage du contrôle.

Il existe trois modes de création de contrôle :

- Les contrôles personnalisés,
- L'héritage de contrôles,
- Les contrôles utilisateurs.

La création de contrôles s'inscrit dans le principe de réutilisation du code. La logique est créée en un seul endroit et peut être utilisée plusieurs fois. L'avantage est d'autant plus important pour la maintenance d'application car pour changer le comportement de ce contrôle, il n'y aura qu'un fichier à modifier.

Les contrôles personnalisés

Ces contrôles offrent les plus grandes possibilités de personnalisation tant au niveau graphique que logique. Un contrôle personnalisé hérite directement de la classe `Control`. Il est donc nécessaire d'écrire toute la logique d'affichage ce qui, suivant le résultat attendu, peut être une phase très longue et compliquée. Les méthodes, propriétés et évènements doivent également être définis par le développeur.

La classe de base `Control` expose l'évènement `Paint`. C'est celui-ci qui est levé lorsque le contrôle est généré et cela implique l'exécution du gestionnaire de l'évènement par défaut `OnPaint`. Cette méthode reçoit un paramètre unique du type `PaintEventArgs` contenant les informations requises sur la surface de dessin du contrôle. Le type `PaintEventArgs` possède deux propriétés, `Graphics` du type `System.Drawing.Graphics` et `ClipRectangle` du type `System.Drawing.Rectangle`. Pour ajouter la logique de dessin au contrôle, il faut surcharger la méthode `OnPaint` et y ajouter le code de dessin :

```
protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
{
    // Code de dessin du contrôle
}
```

La propriété `Graphics` de l'objet `PaintEventArgs` représente la surface du contrôle tandis que la propriété `ClipRectangle` représente la zone devant être dessinée. Lors de la première représentation du contrôle, la propriété `ClipRectangle` représente les limites du contrôle. Ces limites peuvent ensuite être modifiées, par exemple si un contrôle au dessus en cache une partie de telle sorte que le contrôle ait besoin d'être redessiné. La partie `ClipRectangle` représentera la région à modifier.

Créez un dossier **Controls** à la racine du projet et ajoutez une nouvelle classe nommée **CustomControl** définie de la manière suivante :

```
using System.Drawing;

namespace SelfMailer.Controls
{
    public class CustomControl : System.Windows.Forms.Control
    {
        protected override void OnPaint
                        (System.Windows.Forms.PaintEventArgs e)
        {
            Rectangle R = new Rectangle(0, 0,
                            this.Size.Width, this.Size.Height);
            e.Graphics.FillRectangle(Brushes.Green, R);
        }
    }
}
```

Dans le constructeur du formulaire **MailServerSettings**, ajoutez le code d'instanciation du contrôle personnalisé :

```
Controls.CustomControl C = new Controls.CustomControl();
C.Location = new System.Drawing.Point(0, 0);
C.Size = this.Size;
this.Controls.Add(C);
```

Ce code instancie un nouveau contrôle du type `CustomControl`, lui affecte la position en haut à gauche et définit sa taille à celle du formulaire. Pour finir, le contrôle est ajouté à la collection des contrôles du formulaire.

Lancez l'application (**F5**) et ouvrez le formulaire des paramètres de serveur mail pour voir que le contrôle, qui représente un simple rectangle vert, remplit le formulaire comme une couleur de fond.

Les possibilités de dessin avec GDI+ seront abordées plus loin dans cet ouvrage, à la section Le dessin avec GDI+ du chapitre Pour aller plus loin.

Il suffit ensuite d'ajouter les membres requis pour la logique du contrôle afin de le finaliser.

L'héritage de contrôles

Si le but est d'étendre les fonctionnalités d'un contrôle existant, que ce soit un contrôle du Framework .NET ou d'un éditeur tiers, la manière la plus rapide est d'hériter de ce contrôle. Le nouveau contrôle possède ainsi tous les membres et la représentation visuelle de sa classe parente. Il n'y a plus qu'à rajouter la logique de traitement. Au même titre que les contrôles personnalisés, il reste possible de surcharger la méthode `OnPaint` pour modifier l'aspect visuel du contrôle.

Si une application comporte plusieurs formulaires qui requièrent un email comme champ de saisie, il serait préférable de créer un contrôle héritant de la classe `TextBox` et d'y implémenter la logique de validation puis d'ajouter ce contrôle aux formulaires de manière à ne pas répéter le code de validation dans chacun d'eux.

La création d'un contrôle hérité se fait de la même manière qu'un contrôle personnalisé, en créant une classe qui va hériter du contrôle ayant le comportement de base souhaité. Créez la classe `EmailTextBox` dans le dossier **Controls** et faites-la hériter de la classe `TextBox` :

```
public class EmailTextBox : System.Windows.Forms.TextBox
{
}
```

Ajoutez une surcharge de la méthode `OnValidating` pour effectuer les vérifications sur le format et ajoutez le code de la méthode `FromEmail_Validating` du formulaire **MailServerSettings** :

```
protected override void
          OnValidating(System.ComponentModel.CancelEventArgs e)
{
    base.OnValidating(e);
    string pattern = @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\." +
                     @"[0-9]{1,3}\.[0-9]{1,3}\.)|" +
                     @"(([a-zA-Z0-9\-]+\.)+))" +
                     @"([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
    Regex reg = new Regex(pattern);
    if (!reg.IsMatch(this.Text))
    {
         this.BackColor = Color.Bisque;
         e.Cancel = true;
    }
    else
        this.BackColor = this.PreviousBackColor;
}
```

Des modifications sont à apporter car on n'accède plus à la propriété `Text` du contrôle à partir du formulaire. Il faut donc remplacer :

```
this.FromEmail.Text
```

par un accès direct :

```
this.Text
```

La première instruction :

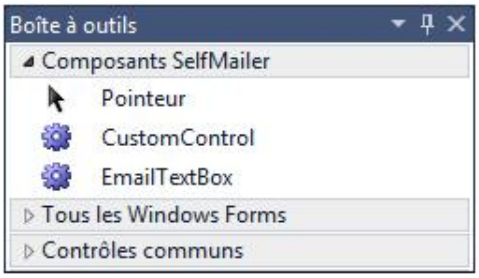
```
base.OnValidating(e);
```

permet d'appeler la méthode de validation de la classe de base. Ainsi, il y a une première validation par la classe de base puis il y a validation du contrôle par le code supplémentaire.

La dernière chose notable est que le composant **ErrorProvider** n'est plus au même niveau. Pour signaler à l'utilisateur que le champ est invalide, au lieu d'afficher une icône, la couleur de fond du contrôle est modifiée. La propriété `PreviousBackColor`, initialisée avec la couleur de fond de base dans le constructeur, permet de la conserver afin de la réaffecter au contrôle en cas de succès de la validation :

```
protected Color PreviousBackColor { get; set; }

public EmailTextBox()
{
    this.PreviousBackColor = this.BackColor;
}
```

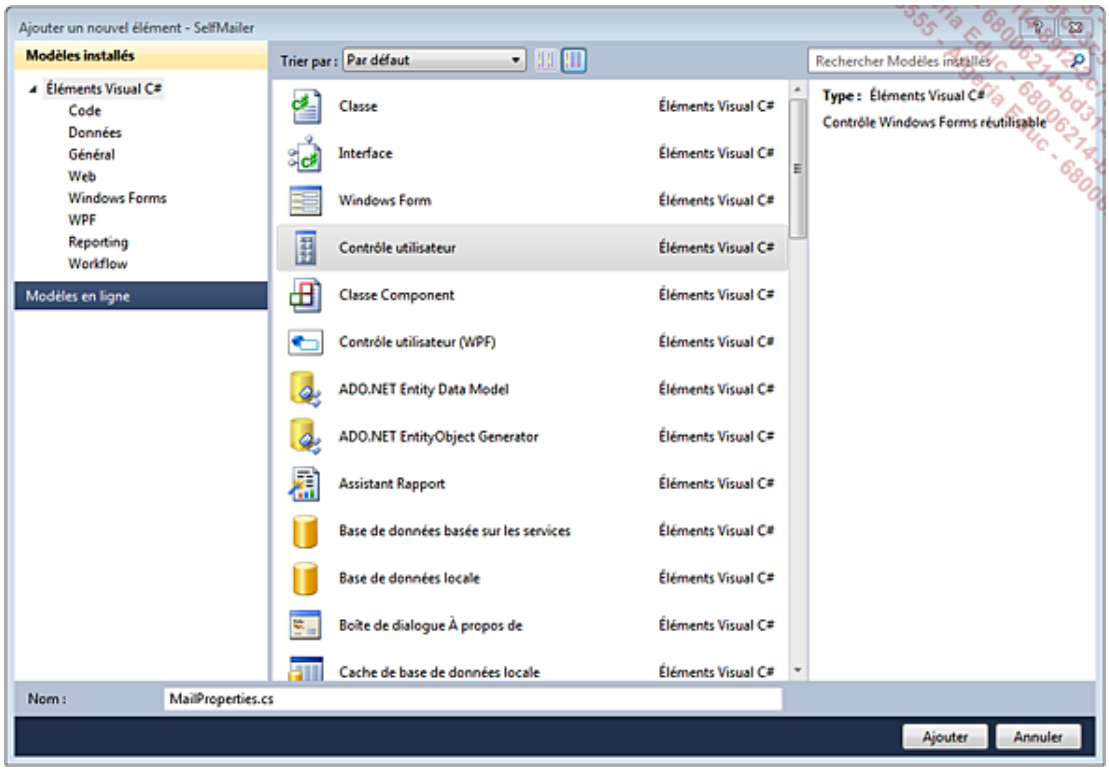
Le gestionnaire d'évènement `FromEmail_Validating` est devenu inutile puisque la validation se fait au sein du contrôle. De plus, vous pouvez supprimer le contrôle de type `TextBox` pour la saisie de l'email de l'expéditeur et le remplacer par un contrôle de type `EmailTextBox` qui a été ajouté par Visual Studio dans la boîte à outils sous le groupe **Composants SelfMailer** (où **SelfMailer** représente le nom du projet) :



Lancez l'application (**F5**) pour tester la fonctionnalité.

Les contrôles utilisateurs

Le but d'un contrôle utilisateur est de regrouper de manière logique des contrôles afin d'obtenir une entité réutilisable. La création se fait par l'ajout au projet d'un **Contrôle utilisateur** depuis la fenêtre d'ajout d'un nouvel élément. Ajoutez un contrôle utilisateur nommé **MailProperties** dans le dossier **Controls** du projet :



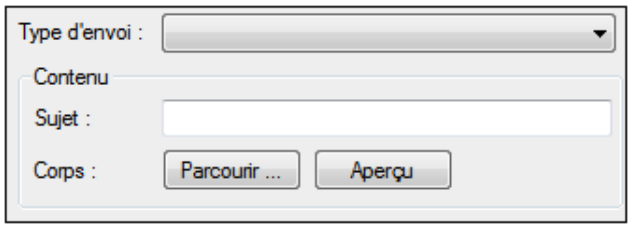
Les contrôles utilisateurs peuvent être construits avec le concepteur de vue au même titre que les formulaires. Ils agissent à la manière d'un conteneur. Il suffit donc de faire glisser les contrôles depuis la boîte à outils sur le concepteur de vue pour ajouter des contrôles.

Ajoutez les contrôles suivants au contrôle utilisateur **MailProperties** :

Type	Propriété	Valeur
`Label`	`Name`	lblSendType
	`Text`	Type d'envoi :
`ComboBox`	`Name`	SendType
	`DropDownStyle`	DropDownList
`GroupBox`	`Name`	MailContent
	`Anchor`	Top, Left, Right
	`Text`	Contenu
	`Enabled`	False
`Label`	`Name`	lblSubject
	`Text`	Sujet :
`TextBox`	`Name`	Subject

	Anchor	Top, Left, Right
Label	Name	lblBody
	Text	Corps :
Button	Name	LoadBody
	Text	Parcourir ...
Button	Name	PreviewBody
	Text	Aperçu

Le résultat obtenu devrait correspondre à ceci :



Ce nouveau contrôle utilisateur devient disponible dans la boîte à outils et peut être déposé sur n'importe quel formulaire ou sur un autre contrôle utilisateur. Il est également possible d'ajouter le contrôle utilisateur via le code en procédant à son instanciation et en l'ajoutant à la collection de contrôles du conteneur dans lequel le placer.

Ajoutez la méthode suivante au formulaire **Main** :

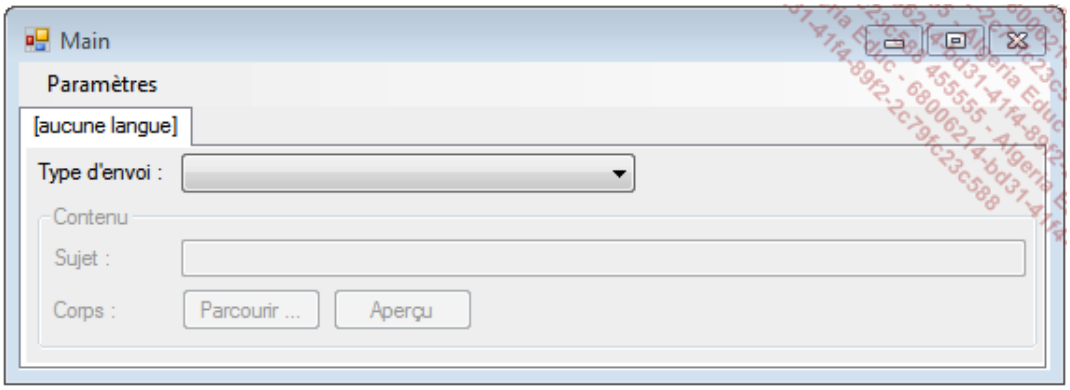
```
private void CreateDefaultTab()
{
    Controls.MailProperties MP = new Controls.MailProperties();
    MP.Name = "mailProperties";
    MP.Dock = DockStyle.Fill;
    TabPage  tp = new TabPage("[aucune langue]");
    tp.Controls.Add(MP);
    this.MainTab.TabPages.Add(tp);
}
```

Cette méthode instancie un nouvel objet de type `MailProperties`. Après l'affection de la propriété `Name` et de sa propriété `Dock`, le contrôle est ajouté à la collection de contrôles d'un nouvel objet de type `TabPage`. L'objet `TabPage` est ensuite ajouté à la collection de pages du contrôle `TabControl`.

Ajoutez, dans le constructeur du formulaire, un appel à cette méthode :

```
public Main()
{
    InitializeComponent();
    this.CreateDefaultTab();
}
```

Lancez l'application (**F5**) pour observer le résultat :



Tous les contrôles possèdent une propriété `Modifiers` permettant de définir de quelle manière les propriétés peuvent être accédées et surtout par qui. Par défaut les contrôles ont la propriété `Modifiers` à la valeur `Private`. Cela signifie qu'ils ne sont accessibles que depuis le formulaire ou le contrôle utilisateur qui les hébergent. Pour autoriser d'autres parties de code à modifier les propriétés d'un contrôle enfant d'un formulaire ou d'un contrôle utilisateur, il faut soit modifier cette propriété pour exposer tout le contrôle, avec les valeurs `Public` ou `Internal`, soit déclarer des propriétés pour exposer seulement les membres désirés.

Pour exposer la propriété `Text` de l'objet de type `TextBox` nommé `Subject`, il faudra créer une propriété dans le contrôle utilisateur :

```
internal string MailSubject
{
    get { return this.Subject.Text; }
    set { this.Subject.Text = value; }
}
```

Il est ainsi possible de définir la propriété `Text` depuis un objet contrôle utilisateur :

```
Controls.MailProperties MP = new Controls.MailProperties();
MP.MailSubject = "Mon sujet";
```

Les valeurs saisies dans le contrôle utilisateur doivent pouvoir être sauvées dans le membre `Project` de la classe `Program`.

Créez une nouvelle classe **MailProperties** dans le dossier **Library**. Cette classe devra implémenter les interfaces `IReportChange` et `IKey` :

```
public class MailProperties : IReportChange, IKey
{
}
```

Ajoutez les membres requis à la classe `MailProperty` :

```
protected string name;
protected string sendType;
protected string subject;
protected string body;
protected bool hasChanged;

public string Name
{
    get { return this.name; }
    set
    {
        if (this.name != value)
        {
            this.name = value;
            this.HasChanged = true;
        }
    }
}
public string SendType
{
    get { return this.sendType; }
    set
```

```
    {
        if (this.sendType != value)
        {
            this.sendType = value;
            this.HasChanged = true;
        }
    }
}
public string Subject
{
    get { return this.subject; }
    set
    {
        if (this.subject != value)
        {
             this.subject = value;
            this.HasChanged = true;
        }
    }
}
public string Body
{
    get { return this.body; }
    set
    {
        if (this.body != value)
        {
            this.body = value;
            this.HasChanged = true;
        }
    }
}
public bool HasChanged
{
    get { return hasChanged; }
    set
    {
        if (this.hasChanged != value)
        {
            this.hasChanged = value;
            if (this.Changed != null)
                this.Changed(this,
                        new ChangedEventArgs(this.HasChanged));
        }
    }
}
public string Key
{
    get { return this.name; }
}

public event EventHandler<ChangedEventArgs> Changed;
```

Étant donné que l'application pourra contenir plusieurs contrôles **MailProperties**, les données inhérentes devront être stockées sous forme de liste dans la classe `Project`.

Ajoutez le membre `MailProperties` à la classe `Project` :

```
public ReportChangeList<MailProperties> MailProperties
{ get; set; }
```

Initialisez le membre `MailProperties` dans le constructeur de la classe `Project` :

```
this.MailProperties = new ReportChangeList<MailProperties>();
this.MailProperties.Changed +=
           new EventHandler<ChangedEventArgs>(ChildChanged);
```

Il ne reste plus qu'à faire la liaison entre les objets et les évènements du formulaire. Cette étape ne sera pas détaillée ici. Les sources du projet contiennent ces fonctionnalités et d'autres formulaires qui seront utilisés pour la suite de l'ouvrage.

Introduction

WPF (*Windows Presentation Foundation*) est une suite de bibliothèques permettant le développement d'interfaces graphiques. Ces bibliothèques sont intégrées au Framework .NET depuis la version 3.0 et elles sont basées sur les classes et le CLR du Framework .NET 2.0. En plus du développement d'applications, WPF est également la base pour des développements Silverlight ou Surface.

WPF fait suite à la bibliothèque de formulaires Windows. Les fonctionnalités et les techniques de développement sont différentes. On peut considérer que WPF complète les formulaires Windows afin de faciliter la conception d'interfaces graphiques en intégrant des fonctionnalités de graphismes vectoriels, d'animations ou encore d'effets sur les éléments (floutage, ombrage...) qui sont plus faciles à intégrer et à maintenir qu'avec les formulaires Windows utilisant GDI+.

Les outils de développement

Depuis la version 2008, Visual Studio supporte le développement WPF. Par analogie au développement de formulaires Windows, un formulaire WPF comporte un fichier de code contenant la logique et un fichier de présentation au format XAML (*eXtensible Application Markup Language*).

Expression Blend est spécialisé dans la création et l'édition d'interfaces graphiques pour les applications WPF. Son interface est plus conviviale que Visual Studio pour les designers. Les contrôles et les animations peuvent être édités visuellement et il est possible de concevoir des applications sans une seule ligne de code.

Expression Blend peut être utilisé en parallèle avec Visual Studio car il utilise également les solutions Visual Studio. En conclusion, Expression Blend sera utilisé pour la conception de l'interface graphique et Visual Studio pour la logique des formulaires.

Ajoutez un nouveau projet (**Fichier** - **Ajouter** - **Nouveau projet...**) de type **Application WPF** et nommé **IntroductionWPF**. Visual Studio crée un projet de base avec un formulaire par défaut **MainWindow.xaml**.

Le langage XAML

Prononcé *zammel*, le XAML est un langage déclaratif dérivé du XML permettant de séparer les couches présentation et logique dans un formulaire WPF. La création d'une interface graphique revient à créer des objets et à en définir les propriétés. Une page XAML décrit la classe qui sera générée à l'exécution.

La création d'un bouton sur un formulaire Windows s'effectue de la manière suivante :

```
Button MonBouton = new Button()
MonBouton.Name = " MonBouton ";
MonBouton.Location = new System.Drawing.Point(339, 239);
MonBouton.Size = new System.Drawing.Size(75, 23);
MonBouton.Text = "Cliquer ici !";
MonBouton.Click += new System.EventHandler(MonBouton_Click);
```

Voici comment créer le même bouton en XAML :

```
<Button Name="MonBouton"
        Margin="339,239,0,0"
        Width="75"
        Height="23"
        Content="Cliquer ici !"
        Click="MonBouton_Click" />
```

Comme vous pouvez le remarquer, la syntaxe est proche du XML. Le nom de la balise (`Button` dans l'exemple) correspond à une classe et les attributs (`Name`, `Margin`, `Width`...) correspondent aux propriétés de la classe. Les valeurs des propriétés sont toutes de type `string` dans le code XAML. C'est le compilateur qui effectue la conversion dans le type de la propriété. Les évènements et le gestionnaire associé sont également définis avec un attribut. Là encore, le compilateur se charge de l'association entre l'évènement et son gestionnaire.

Le XAML étant dérivé du langage XML, sa syntaxe doit se conformer aux mêmes règles :

- Un fichier doit contenir un seul et unique élément racine.
- Chaque balise ouverte doit être fermée.
- La dernière balise ouverte doit être la première à être fermée.
- Chaque attribut doit avoir une valeur définie entre guillemets simples ou doubles.
- La casse des balises et attributs est prise en compte.

La balise racine du fichier XAML définit également les schémas XSD (*XML Schema Definition*) utilisés qui servent à spécifier :

- Les balises et attributs valides.
- Les valeurs valides pour un attribut.
- Les balises pouvant être imbriquées dans une autre.

Dans le fichier **MainWindow.xaml** du projet **IntroductionWPF**, la balise racine `Window` définit les schémas XSD utilisés dans le document :

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Ces schémas permettent de faire la liaison entre les objets et les classes du Framework .NET. Pour illustrer cette liaison, dans l'exemple précédent, le bouton aurait pu être défini de la manière suivante :

```
<Button Name="MonBouton"
        Margin="339,239,0,0"
        Width="75"
        Height="23"
```

```
        Click="MonBouton_Click">
    Cliquer ici !
</Button>
```

Le texte du bouton n'est plus défini dans un attribut `Content` mais en tant que contenu de la balise `Button`. Cette syntaxe est valide car la classe `Button` dérive de la classe `ContentControl` qui est marquée avec l'attribut `[ContentProperty("Content")]`. Cela signifie que le contenu de la balise correspond à la propriété `Content` (la propriété `Content` ne peut pas être définie dans le contenu de la balise et dans un attribut en même temps).

Une première application WPF

Une application WPF complexe sera composée d'une partie XAML et d'une autre partie de code étant donné que le XAML ne permet pas de définir la logique d'une application.

En observant le projet **IntroductionWPF**, vous remarquez que Visual Studio a créé le projet avec deux fichiers XAML :

- **App.xaml** : le XAML suit les règles de base du XML qui spécifie qu'un fichier ne doit avoir qu'un seul et unique élément racine. Pour cette raison, l'objet `Application` doit être défini à part de l'objet `Window` :

```
<Application x:Class="IntroductionWPF.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

  Les attributs de la balise racine `Application` permettent de définir la classe liée au fichier XAML (`x:Class`), les espaces de noms XML (`xmlns` et `xmlns:x`) ainsi que le formulaire de démarrage (`StartupUri`).

- **MainWindow.xaml** : le formulaire WPF est défini dans ce fichier. L'élément racine est une balise `Window` :

```
<Window x:Class="IntroductionWPF.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

  La balise `Window` définit la classe liée au fichier XAML (`x:Class`), les espaces de noms XML (`xmlns` et `xmlns:x`) ainsi que les propriétés du formulaire (`Title`, `Height` et `Width`).

Dans ces deux fichiers, l'élément racine définit des espaces de noms. Ils servent à organiser les balises XML. Deux balises peuvent porter le même nom et se trouver dans deux espaces de noms différents. L'espace de noms par défaut est indiqué sans alias :

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Cette valeur est définie par Microsoft et représente l'espace de noms qui contient la plupart des balises XAML. Pour les autres espaces de noms, il faut spécifier un alias. La seconde déclaration d'espace de noms des fichiers utilise l'alias `x` :

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Cet espace de noms contient des éléments supplémentaires accessibles en utilisant le préfixe `x:`, l'attribut `Class` des balises racines des deux fichiers en est un exemple :

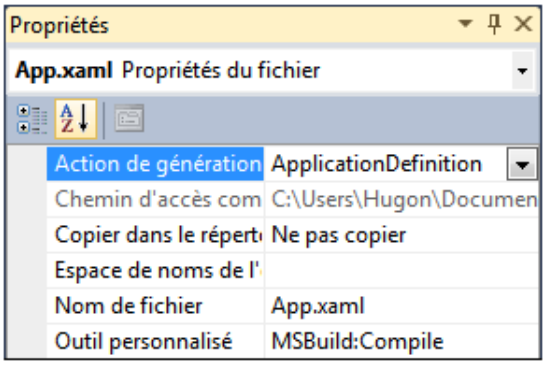
```
x:Class="IntroductionWPF.MainWindow"
```

Ce préfixe peut également être utilisé pour les balises :

```
<x:Code></x:Code>
```

Contrairement à une application Windows, une application WPF ne contient pas de méthode `Main` (point d'entrée de l'application). C'est le compilateur qui va la générer. Le compilateur sait quel fichier XAML contient la définition de l'application car dans les propriétés de celui-ci, la valeur de la propriété **Action de génération** est définie à **ApplicationDefinition**.

Sélectionnez le fichier **App.xaml** puis ouvrez la fenêtre de propriétés :

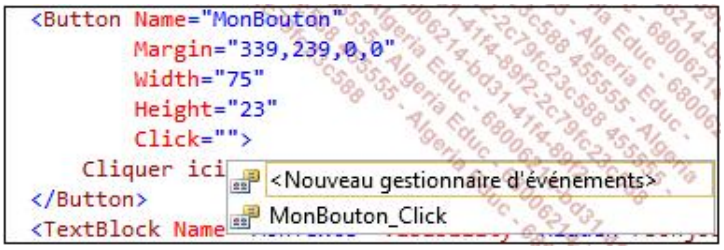


Ajoutez un bouton et un texte sur le formulaire **MainWindow.xaml** dans la balise `Grid` :

```
<TextBlock Name="MonTexte"
           Visibility="Hidden"
           FontSize="32"
           FontWeight="Bold">
    Bonjour !
</TextBlock>
<Button Name="MonBouton"
        Margin="339,239,0,0"
        Width="75"
        Height="23"
        Click="MonBouton_Click">
    Cliquer ici !
</Button>
```

Lorsque l'attribut `Click` est saisi, l'IntelliSense propose la création d'un nouveau gestionnaire d'évènement et insère la méthode dans le fichier de code :

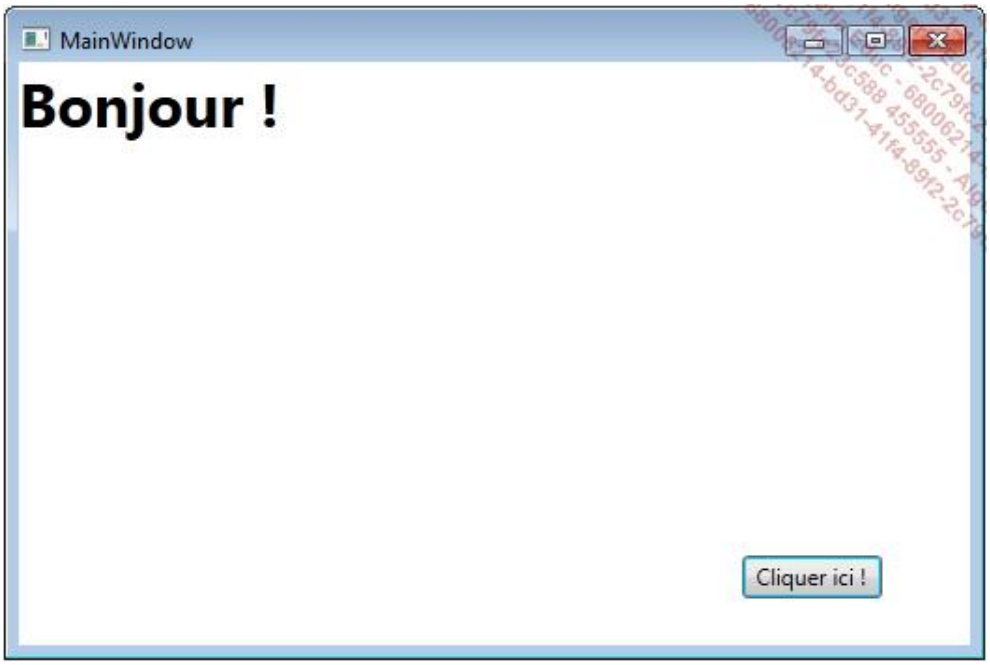
Si le fichier de code contient des méthodes avec des signatures en accord avec l'évènement, l'IntelliSense propose d'en choisir une parmi elles ou d'en créer une nouvelle :



Dans la partie code, ajoutez l'instruction suivante au gestionnaire de l'évènement `Click` de l'objet `MonBouton` :

```
private void MonBouton_Click(object sender, RoutedEventArgs e)
{
        this.MonTexte.Visibility = Visibility.Visible;
}
```

Lancez l'application (dans le menu contextuel du projet, sélectionnez **Déboguer** puis **Démarrer une nouvelle instance**) et cliquez sur le bouton pour faire apparaître le texte :



Les animations

Les animations et leur simplicité de mise en œuvre représentent le réel avantage d'une application WPF. Il existe une variété d'animations disponibles, que ce soit des changements de couleurs, de taille d'objets ou de position. Une animation est une partie d'un objet `StoryBoard` qui peut contenir une ou plusieurs animations s'exécutant de manière séquentielle ou en parallèle.

Pour démarrer automatiquement une animation, vous pouvez inclure l'objet `StoryBoard` dans un objet `BeginStoryBoard` qui sera lié avec la propriété `Triggers` du contrôle déclenchant l'animation. Au lieu de changer la visibilité du texte lors du clic sur le bouton, nous allons créer une animation permettant de modifier son opacité.

La première étape consiste à choisir le bon type d'animation. Dans le cas du changement de l'opacité, une animation de type `DoubleAnimation` conviendra. La valeur de l'opacité devant passer de 0 à 1 en 5 secondes :

```
<DoubleAnimation Storyboard.TargetName="MonTexte"
                 Storyboard.TargetProperty="Opacity"
                 From="0.0" To="1.0" Duration="0:0:5" />
```

L'objet `DoubleAnimation` permet également de spécifier le nom de l'objet cible ainsi que sa propriété qui sera affectée par l'animation.

L'animation doit maintenant être associée à un objet `StoryBoard` et à un évènement de l'objet `MonBouton` :

```
<Button Name="MonBouton"
        Margin="339,239,0,0"
        Width="75"
        Height="23">
    Cliquer ici !
    <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="MonTexte"
                        Storyboard.TargetProperty="Opacity"
                        From="0.0" To="1.0" Duration="0:0:5" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
</Button>
```

Le gestionnaire d'évènement `Click` peut être supprimé du fichier XAML et du code. La propriété `Visibility` de l'objet `MonTexte` doit être supprimée et remplacée par la définition de l'opacité avec la valeur 0 :

```
<TextBlock Name="MonTexte"
           Opacity="0.0"
           FontSize="32"
           FontWeight="Bold">
    Bonjour !
</TextBlock>
```

Lancez l'application et cliquez sur le bouton pour lancer l'animation.

Bien entendu, les animations peuvent être définies dynamiquement dans le code C#. Voici le code permettant de créer la même animation dans la classe `MainWindow` :

```
public partial class MainWindow : Window
{
    private Storyboard monStoryboard;

    public MainWindow()
    {
        InitializeComponent();

        DoubleAnimation monAnimation = new DoubleAnimation();
        monAnimation.From = 0.0;
        monAnimation.To = 1.0;
        monAnimation.Duration =
```

```
                        new Duration(TimeSpan.FromSeconds(5));

        monStoryboard = new Storyboard();
        monStoryboard.Children.Add(monAnimation);
        Storyboard.SetTargetName(monAnimation, MonTexte.Name);
        Storyboard.SetTargetProperty(monAnimation,
                    new PropertyPath(TextBlock.OpacityProperty));

        MonBouton.Click +=
                        new RoutedEventHandler(MonBoutonClick);
    }

    private void MonBoutonClick(object sender, RoutedEventArgs e)
    {
        monStoryboard.Begin(this);
    }
}
```

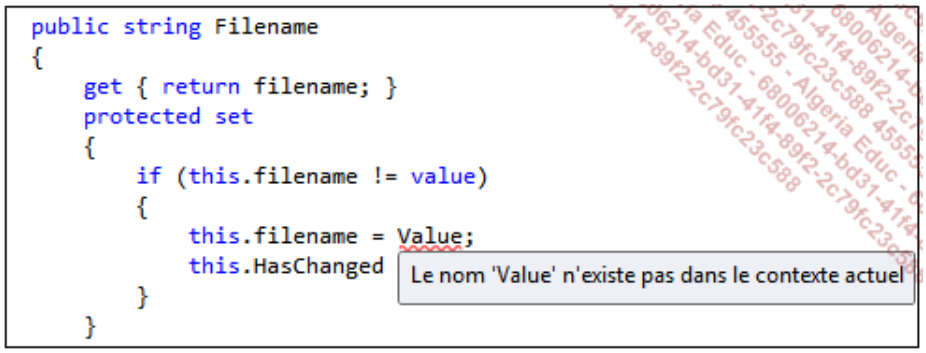
Les types d'erreur

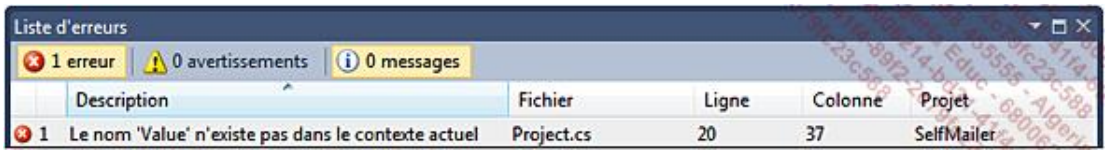
Quelle que soit l'expérience d'un développeur, les erreurs sont inévitables. Le débogage permet de les localiser et de les expliquer afin de pouvoir les corriger. Visual Studio comporte des outils permettant de traquer les erreurs tout au long du cycle de développement d'une application.

Il existe trois types d'erreurs de base lors de la conception d'une application: les erreurs de syntaxe qui correspondent au code que le compilateur ne peut pas interpréter, les erreurs d'exécution qui se produisent lors de l'exécution de l'application et les erreurs de logique qui produisent des résultats non souhaités.

1. Les erreurs de syntaxe

Visual Studio traque les erreurs de syntaxe dans l'éditeur de texte avant même la compilation. Cela peut être une faute de frappe dans le nom d'une variable ou une instruction non valide. Visual Studio facilite l'identification et la localisation de ces erreurs en les soulignant dans le code et en les ajoutant à la fenêtre **Liste d'erreurs** (**Ctrl** + **W**, **E**) :





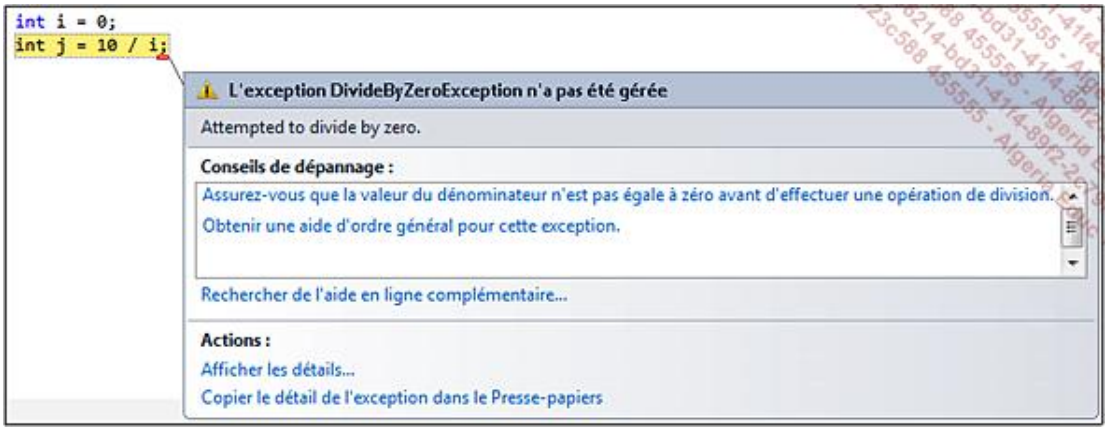
La fenêtre **Liste d'erreurs** énumère toutes les erreurs de l'application en donnant pour chacune d'elles une description, le fichier dans lequel l'instruction se trouve, ainsi que sa ligne. Double cliquer sur une erreur de la liste ouvre le fichier concerné et place le curseur sur l'instruction qui a causé l'erreur. Les erreurs sont classées en trois catégories : erreurs, avertissements et messages. Les erreurs empêchent la compilation de l'application au contraire des avertissements et messages.

2. Les erreurs d'exécution

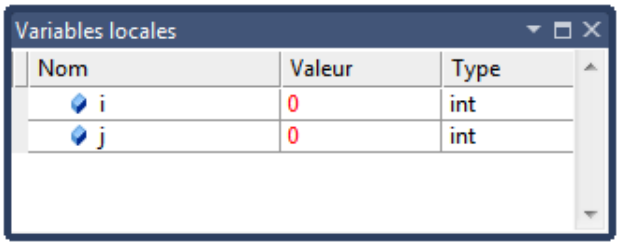
Les erreurs d'exécution se produisent lorsque l'application tente d'effectuer une opération invalide compte tenu de la valeur des variables et du contexte d'exécution. Par exemple, la division par zéro :

```
int i = 0;
int j = 10 / i;
```

La compilation s'effectue avec succès mais lors de l'exécution une exception est levée car la division par 0 est interdite :



Dès la levée d'une exception non gérée, l'exécution de l'application est interrompue et Visual Studio affiche une description de l'erreur. La fenêtre **Variables locales** permet de vérifier la valeur des variables au moment de l'erreur :



Il est possible de modifier ces valeurs puis de reprendre l'exécution de l'application. En cliquant sur la valeur de la variable `i` dans la fenêtre **Variables locales** et en la modifiant par la valeur 2, on peut reprendre l'exécution de l'application (**F5**). L'exécution ne lève plus l'erreur et la valeur 5 est assignée à la variable `j`.

La modification des valeurs des variables locales n'a pour but que de réaliser des essais pour identifier une erreur complexe. Le code d'origine n'est en aucun cas modifié.

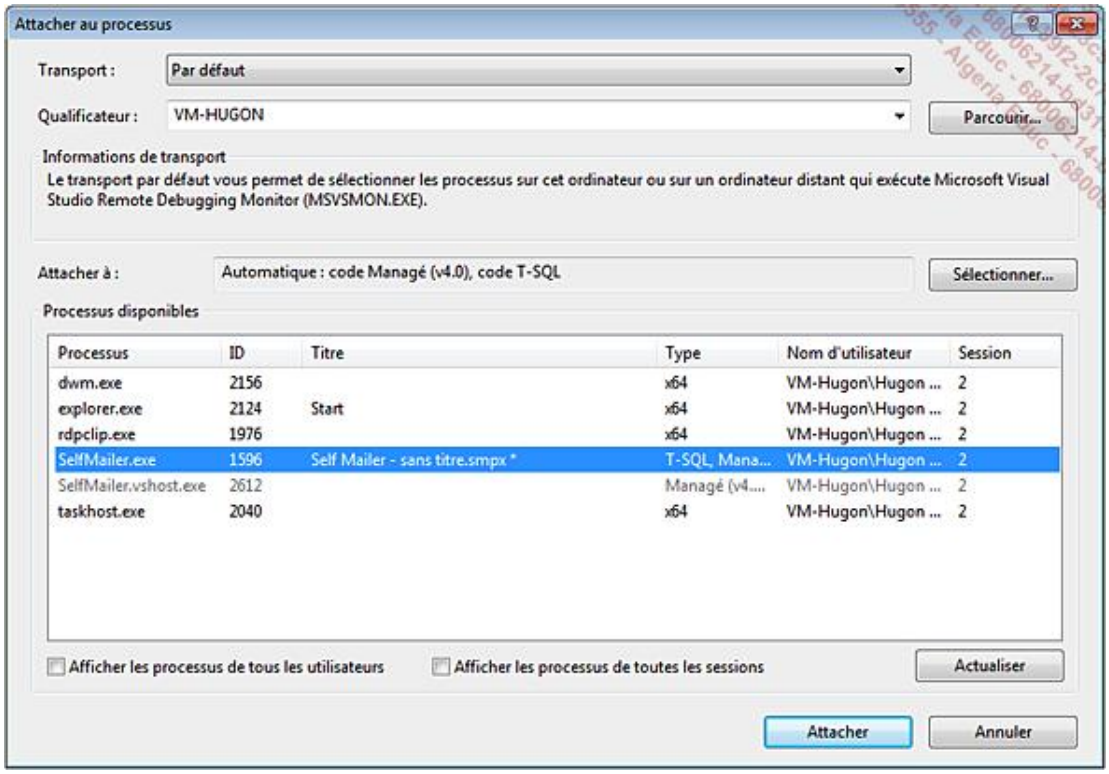
3. Les erreurs de logique

L'application s'exécute sans erreurs et pourtant le résultat n'est pas celui attendu. Les erreurs de logique sont les plus difficiles à détecter et à corriger car vous n'aurez aucune indication sur son origine. Visual Studio met tout de même des outils à disposition pour aider à la localisation de ces erreurs avec notamment le débogueur et les tests unitaires.

Le débogueur

Le débogueur est un outil de Visual Studio qui peut s'attacher à un processus afin de le contrôler. Lorsqu'une application est lancée avec la commande **Déboguer** - **Démarrer le débogage** (**F5**) depuis Visual Studio, l'application est exécutée dans un nouveau processus et Visual Studio s'attache à celui-ci.

Il est possible de lancer l'exécution d'une application depuis Visual Studio sans débogage depuis le menu **Déboguer** - **Exécuter sans débogage** (**Ctrl** + **F5**). La fenêtre **Attacher au processus** (**Déboguer** - **Attacher au processus...**) permet de choisir un processus qui s'exécute et de spécifier à Visual Studio de s'attacher à celui-ci afin d'en contrôler son exécution :



1. Contrôler l'exécution

Le débogueur intégré de Visual Studio permet de contrôler l'exécution d'une application, d'examiner et modifier les valeurs des variables. Le contrôle de l'exécution est passé à Visual Studio lorsque le processus est suspendu (**Ctrl** + **Alt** + **Pause**), qu'un point d'arrêt a été atteint ou qu'une exception a été levée. Le tableau suivant énumère les fonctions du débogueur :

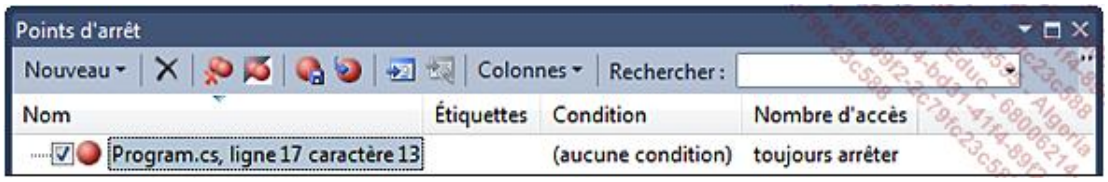
Menu	Description
Démarrer/Continuer le débogage (**F5**)	Lance l'exécution de l'application ou permet de la continuer en mode arrêt.
Interrompre tout (**Ctrl** + **Alt** + **Pause**)	Interrompt l'exécution de l'application et passe en mode arrêt.
Arrêter le débogage (**Shift** + **F5**)	Arrête l'exécution de l'application.
Détacher tout	Détache le débogueur des processus. Visual Studio repasse en mode conception mais l'application continue de s'exécuter.
Redémarrer (**Ctrl** + **Shift** + **F5**)	Redémarre l'application.
Attacher au processus...	Affiche la fenêtre **Attacher au processus**.

Exceptions... (**Ctrl** + **D**, **E**)	Affiche la fenêtre **Exceptions** permettant de configurer les types d'erreurs qui entraîneront le mode arrêt.
Pas à pas détaillé (**F11**)	En mode arrêt, permet d'exécuter l'instruction suivante. Si l'instruction est un appel de méthode, l'exécution s'arrête au début de celle-ci.
Pas à pas principal (**F10**)	En mode arrêt, permet d'exécuter l'instruction suivante sans entrer dans une méthode appelée.
Pas à pas sortant (**Shift** + **F11**)	Termine l'exécution de la méthode en cours et s'arrête à l'instruction suivante de la méthode appelante.
Espion express... (**Ctrl** + **D**, **Q**)	Affiche la fenêtre **Espion express** permettant de parcourir les variables de l'application et de les ajouter à la fenêtre **Espion**.

Les autres éléments du menu déboguer seront détaillés dans les sections suivantes.

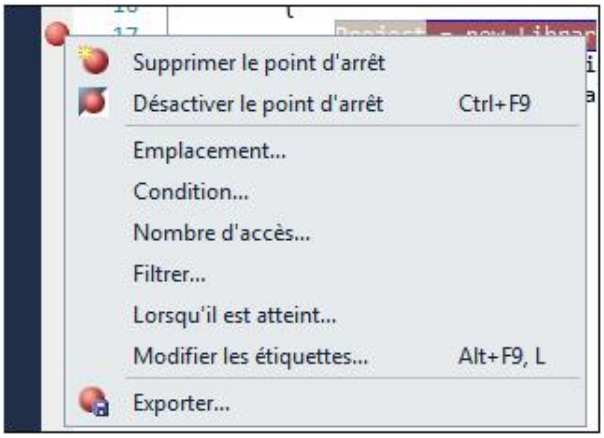
2. Les points d'arrêt

La fenêtre **Points d'arrêt** est disponible depuis le menu **Déboguer** - **Fenêtres** - **Points d'arrêt** (**Ctrl** + **D**, **B**). Elle permet de gérer les points d'arrêt de l'application :



Pour ajouter un point d'arrêt sur une instruction, il suffit soit de placer le curseur sur celle-ci puis d'appuyer sur la touche **F9** ou de cliquer dans la colonne de point d'arrêt à gauche :

Un point d'arrêt peut également être géré directement dans l'éditeur de texte en ouvrant le menu contextuel avec un clic du bouton droit de la souris sur le point d'arrêt :

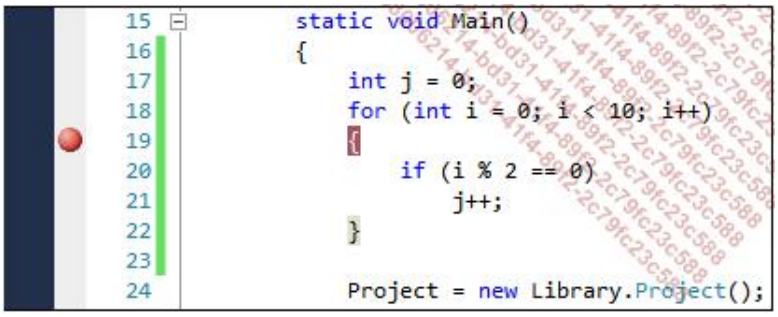


a. Les conditions d'arrêt

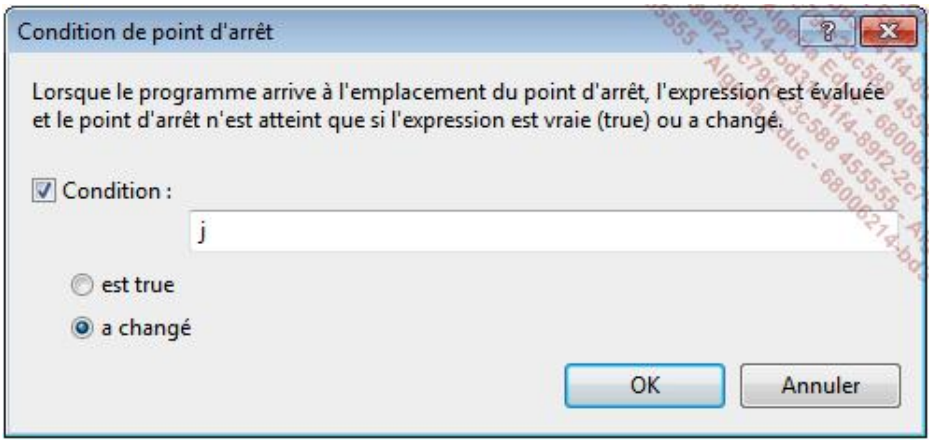
Par défaut, un point d'arrêt spécifie que le débogueur doit se mettre en mode arrêt dès que l'exécution de

l'application arrive sur l'instruction. L'arrêt se fait avant que l'instruction du point d'arrêt ne soit exécutée. La prochaine instruction qui sera exécutée est alors surlignée en jaune :

Il est possible de définir des conditions à un point d'arrêt. Le menu **Condition...** ouvre la fenêtre **Condition de point d'arrêt** permettant de spécifier dans quel cas le débogueur va se mettre en mode arrêt lorsque l'exécution atteindra le point d'arrêt. Prenons l'exemple suivant :



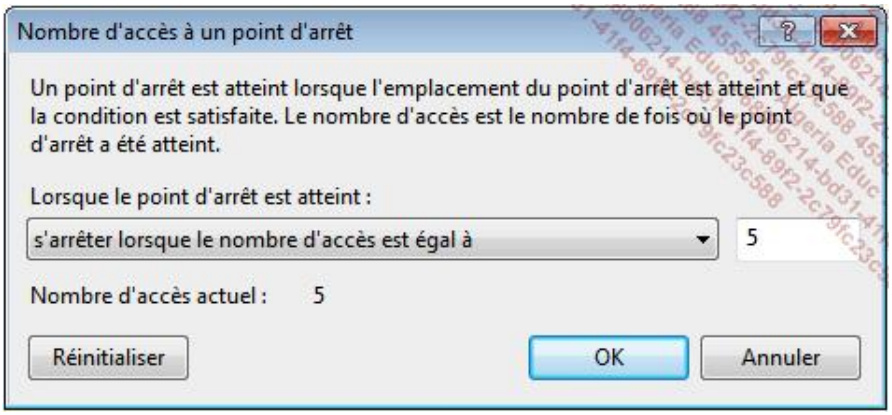
Pour que le débogueur passe en mode arrêt seulement si la variable j est modifiée, la fenêtre **Condition de point d'arrêt** doit être définie de la manière suivante :



Vous remarquerez que les variables de l'application peuvent être utilisées pour définir une condition.

b. Le nombre d'accès

Le menu **Nombre d'accès...** ouvre la fenêtre **Nombre d'accès à un point d'arrêt** :



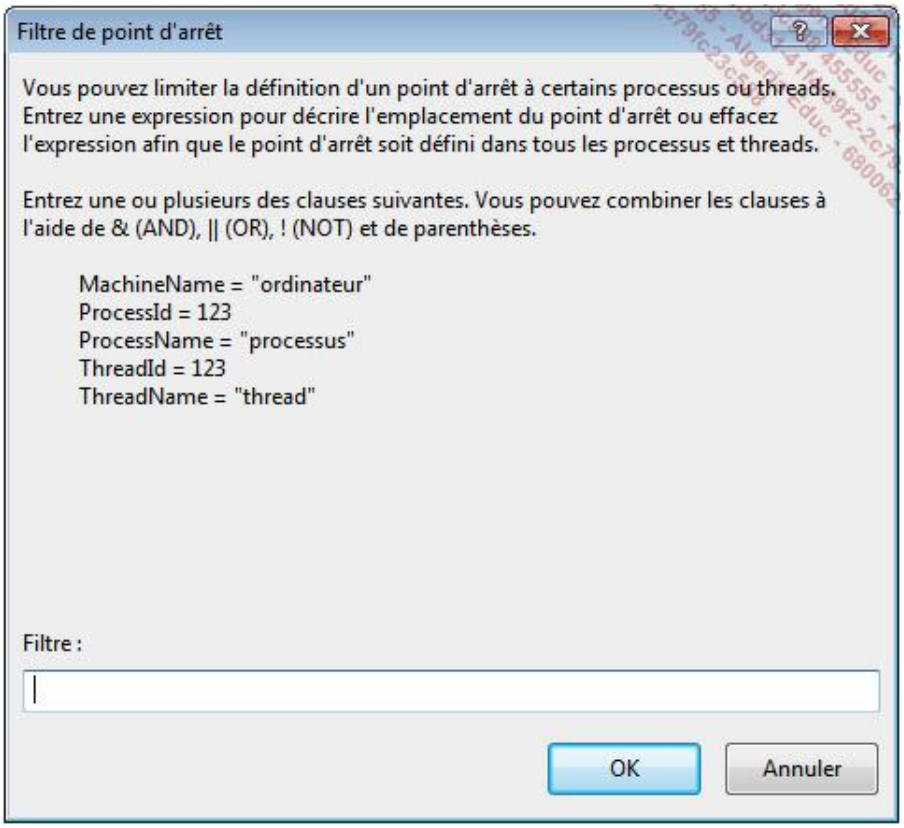
Chaque fois que le débogueur atteint un point d'arrêt, le compteur interne de celui-ci est incrémenté. Il est donc possible de spécifier si le mode arrêt doit toujours se déclencher ou si le compteur doit satisfaire une condition :

- S'arrêter lorsque le nombre d'accès est égal à N.
- S'arrêter lorsque le nombre d'accès est un multiple de N.
- S'arrêter lorsque le nombre d'accès est supérieur ou égal à N.

Le bouton **Réinitialiser** permet une remise à zéro du compteur en cours d'exécution de l'application.

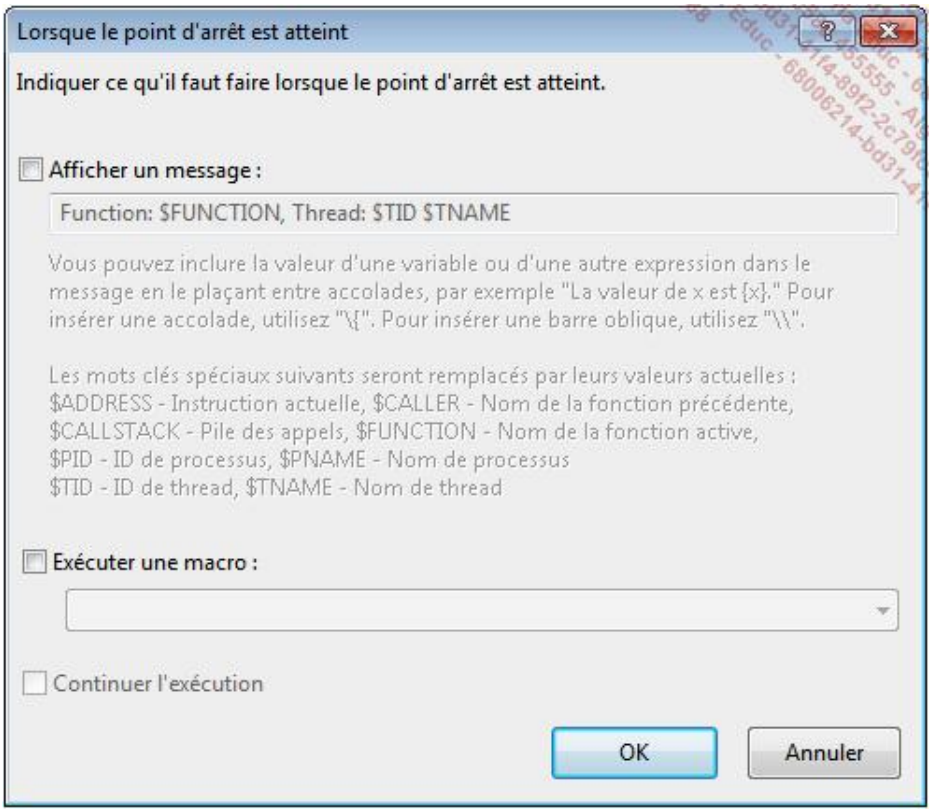
c. Le filtrage

Le menu **Filtrer...** ouvre la fenêtre **Filtre de point d'arrêt** permettant de limiter le passage en mode arrêt en fonction de conditions liées à la machine qui exécute l'application ou en fonction du processus de l'application :



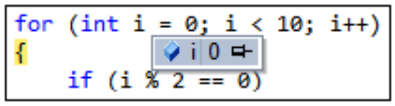
d. L'exécution de commande

Le menu **Lorsqu'il est atteint...** ouvre la fenêtre **Lorsque le point d'arrêt est atteint** afin de choisir parmi l'affichage d'un message et/ou l'exécution d'une macro lorsque le point d'arrêt est atteint lors de l'exécution :

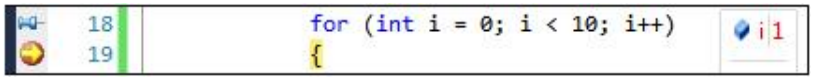


3. Les DataTips

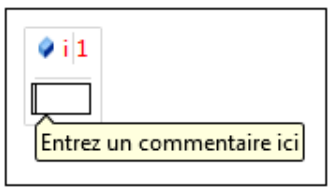
Les DataTips sont une nouveauté de Visual Studio 2010. Ils permettent de mieux traquer les variables et les expressions dans le débogueur. Pour ajouter un DataTip, il faut survoler une variable ou sélectionner une expression et cliquer sur l'icône épingle à droite de la valeur affichée par l'IntelliSense :



Lorsqu'un DataTip est défini, une épingle apparaît à gauche du numéro de ligne et l'IntelliSense est toujours visible pour la valeur :

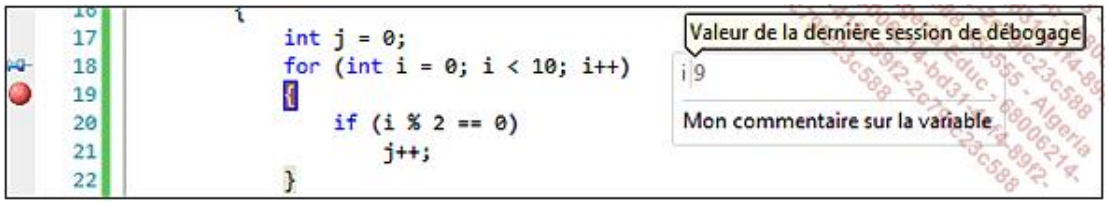


En survolant le DataTip, vous pouvez spécifier et afficher un commentaire :



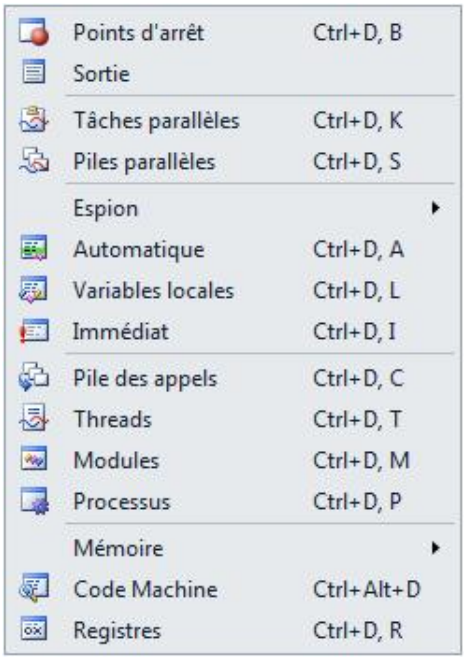
Vous pouvez épingler autant de DataTips que vous le souhaitez. Cela peut être des variables de premier niveau, comme dans l'exemple précédent, ou cela peut être des variables d'un objet.

L’aspect le plus intéressant des DataTips est qu’ils sont accessibles depuis Visual Studio que ce soit en mode débogage ou en mode conception. Lorsque vous survolez une épingle dans l’éditeur de texte, la valeur de la dernière session de débogage s’affiche :



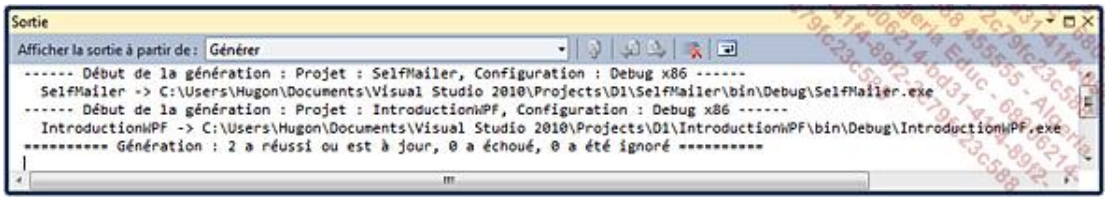
Les fenêtres

Le menu **Déboguer** - **Fenêtres** contient une multitude d'outils pour faciliter le débogage du code. Certaines d'entre elles sont disponibles en mode conception et débogage mais la plupart ne seront accessibles qu'en mode débogage :



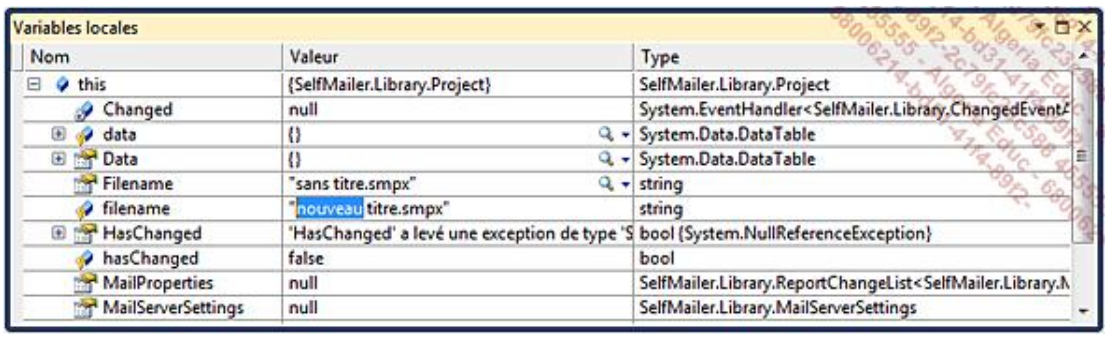
1. La fenêtre Sortie

La fenêtre **Sortie** affiche l'ensemble des informations de compilation et d'exécution de l'application. Cela comprend la génération de l'application ainsi que la sortie des instructions `Debug` et `Trace` :



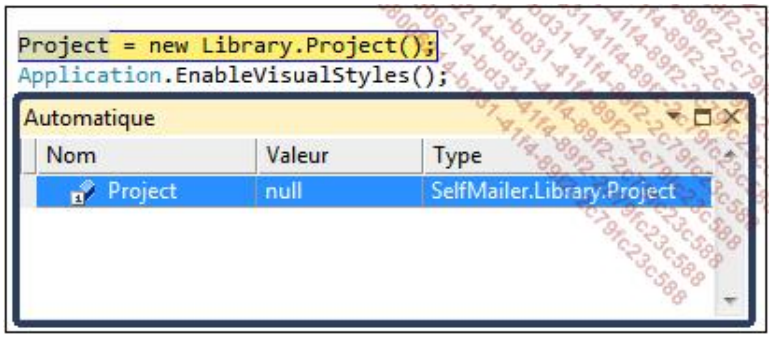
2. La fenêtre Variables locales

La fenêtre **Variables locales** (**Ctrl** + **D**, **L**) affiche les variables de la méthode en cours. Les variables sont présentées sous forme de tableau avec leur nom, valeur et type. Pour les objets complexes, le signe + à gauche du nom de la variable permet d'étendre l'arbre des membres pour afficher leur valeur. La valeur d'une variable apparaît en rouge lorsqu'elle vient d'être modifiée soit lors de l'exécution de l'application ou à partir de la fenêtre elle-même :



3. La fenêtre Automatique

La fenêtre **Automatique** (**Ctrl** + **D**, **A**) ressemble à la fenêtre **Variables locales** à la différence que seules les variables des instructions courante et précédente sont affichées :

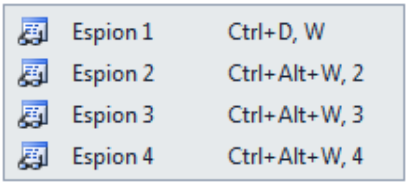


4. La fenêtre Espion

La fenêtre **Espion** (**Ctrl** + **D**, **W**) se présente de la même manière que les fenêtres **Variables locales** et **Automatique**, elle permet de surveiller des variables de son choix même si elles sont hors de portée.

Pour ajouter une variable à cette fenêtre, sélectionnez une variable ou une expression et ouvrez le menu contextuel. Choisissez l'option **Ajouter un espion**.

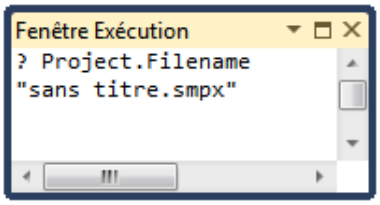
Visual Studio offre quatre fenêtres **Espion** permettant d'organiser les variables en plusieurs jeux :



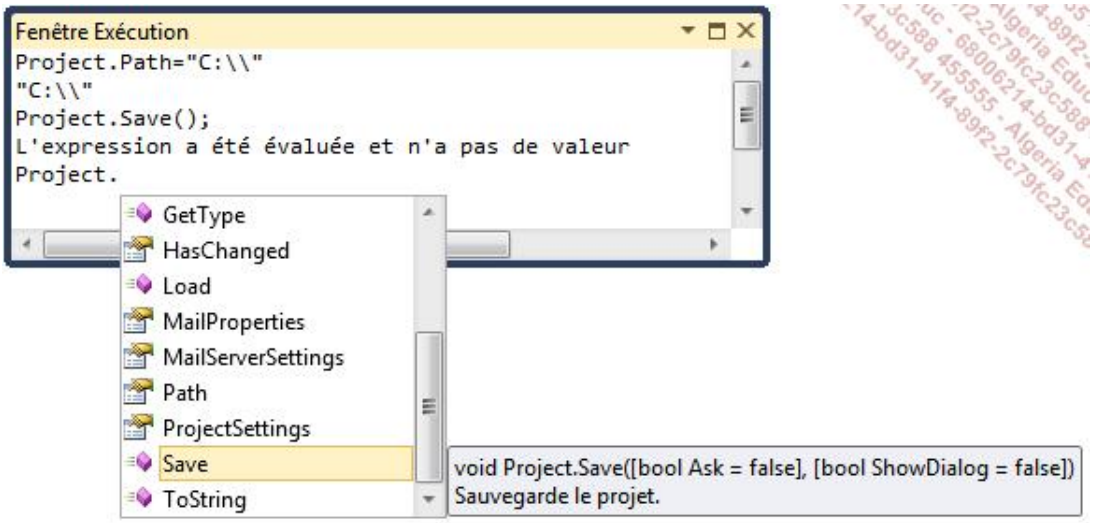
5. La fenêtre Exécution

La fenêtre **Exécution** (**Ctrl** + **D**, **I**) nommée **Immédiat** dans le menu **Déboguer** - **Fenêtres** permet d'exécuter des instructions, d'évaluer des expressions ou de modifier la valeur des variables sous la forme de ligne de commande.

Pour afficher la valeur d'une variable, commencez l'instruction avec le caractère ? :



Pour exécuter une méthode ou modifier la valeur d'une variable, tapez l'instruction directement dans la fenêtre **Exécution**. L'IntelliSense est disponible dans cette fenêtre mais les erreurs de syntaxe ne sont pas signalées :



6. Les autres fenêtres

Le tableau ci-dessous énumère les autres fenêtres des outils disponibles dans le débogueur de Visual Studio :

Fenêtre	Description
Tâches parallèles (**Ctrl** + **D**, **K**)	Affiche le détail des tâches en cours d'exécution et dans quel thread elles sont traitées.
Piles parallèles (**Ctrl** + **D**, **S**)	Affiche sous forme d'arbre la pile des appels des différents threads de l'application.
Pile des appels (**Ctrl** + **D**, **C**)	Affiche la pile des appels des méthodes.
Threads (**Ctrl** + **D**, **T**)	Affiche et permet de contrôler les threads.
Modules (**Ctrl** + **D**, **M**)	Affiche la liste des modules (DLL et EXE) utilisés par l'application.
Processus (**Ctrl** + **D**, **P**)	Affiche les processus auxquels le débogueur est attaché.
Mémoire	Affiche les valeurs stockées en mémoire sous forme hexadécimale.
Code Machine (**Ctrl** + **Alt** + **D**)	Affiche le code machine créé par le compilateur.
Registres (**Ctrl** + **D**, **R**)	Affiche le contenu du registre.

La classe Exception

Toute application, même testée et déboguée, peut rencontrer des erreurs d'exécution. Par exemple, si l'application a besoin d'une ressource sur un réseau et qu'elle n'est pas disponible, l'erreur doit être interceptée et traitée. Les erreurs ne sont pas seulement dues au code de l'application. Il faut également prendre en considération l'environnement d'exécution pour gérer correctement les erreurs.

Le Framework .NET fournit toute une hiérarchie de classes représentant les différents types d'erreurs pouvant survenir. Toutes ces classes dérivent à un niveau plus ou moins lointain de la classe de base `Exception` qui contient des informations sur l'erreur qui a été levée comme le message, la source, la trace, l'erreur d'origine. La classe `Exception` étant très générale, vous ne souhaiterez pas lever des erreurs de ce type. Deux classes en dérivent permettant de séparer les types d'erreurs :

- `System.SystemException` : cette classe est utilisée pour les erreurs habituellement levées par le Framework .NET. Des classes plus spécifiques en dérivent comme `System.ArgumentException` lorsqu'une méthode comporte des arguments invalides, ou `System.UnauthorizedAccessException` lorsqu'un problème de droit apparaît.

- `System.ApplicationException` : cette classe sert de classe de base pour créer des exceptions personnalisées propres à l'application. Elle est identique à la classe `System.SystemException`, son but est de pouvoir séparer et donc de mieux distinguer la source des erreurs.

À partir du moment où une erreur survient dans l'application, un objet correspondant au type de l'erreur est instancié, la méthode en cours stoppe son exécution et l'exception est remontée à la méthode appelante via la pile des appels. L'exception remonte la pile des appels jusqu'à ce qu'elle rencontre un gestionnaire d'erreurs. Si l'application n'en contient pas, un gestionnaire par défaut gère l'exception affichant un message à l'utilisateur et fermant l'application sans possibilité d'enregistrer les données. Il s'agit d'une erreur fatale. En définissant des gestionnaires d'erreurs, l'application peut gérer plus élégamment les exceptions et suivant le cas reprendre son exécution.

Une exception ne doit néanmoins pas être utilisée dans le but de communiquer entre les objets. Les évènements sont chargés de ça. Une exception doit être utilisée seulement lorsque la poursuite de l'application nécessite une intervention ou une opération spécifique.

La création d’exceptions personnalisées

Les types d’exceptions fournis par le Framework .NET ne correspondent pas forcément au besoin d’une application qui nécessiterait un type d’exception contenant des champs spécifiques. La création d’exceptions personnalisées se fait en créant un nouveau type héritant de la classe `System.ApplicationException`.

Créez une nouvelle classe nommée `ProjectException` dans le dossier **Library** du projet. Cette nouvelle classe dérive du type `System. ApplicationException` et contient une propriété permettant de stocker le projet en cours lors du déclenchement de l’erreur :

```
public class ProjectException : ApplicationException
{
    public Project Project { get; protected set; }

    public ProjectException(Project project)
        : base()
    {
        this.Project = project;
    }
    public ProjectException(Project project, string message)
        : base(message)
    {
        this.Project = project;
    }
    public ProjectException(Project project, string message,
Exception innerException)
        : base(message, innerException)
    {
        this.Project = project;
    }
}
```

Les constructeurs de la classe font appels aux constructeurs de la classe de base pour instancier les membres de base et initialiser la propriété `Project` de la classe.

Le déclenchement des exceptions

En règle générale, une exception est déclenchée par le système mais vous pouvez avoir besoin de déclencher une exception plus spécifique ou plus détaillée pour la traiter plus haut dans la pile des appels.

Observez l'accesseur set de la propriété `SendDelay` de la classe `ProjectSettings` :

```
public int SendDelay
{
    get { return this.sendDelay; }
    set
    {
        if (value < 0)
            throw new ArgumentException("La valeur doit être
supérieure ou égale à 0.", "Délai d'envoi");
        if (this.sendDelay != value)
        {
            this.sendDelay = value;
            this.HasChanged = true;
        }
    }
}
```

Lors de l'affection d'une valeur, celle-ci est testée afin de déterminer si elle est supérieure ou égale à zéro. Le délai d'envoi correspond au temps d'attente entre deux envois d'email. Ce nombre doit donc être strictement positif et si ce n'est pas le cas une exception du type `ArgumentException` est levée :

```
throw new ArgumentException("La valeur doit être supérieure ou
égale à 0.", "Délai d'envoi");
```

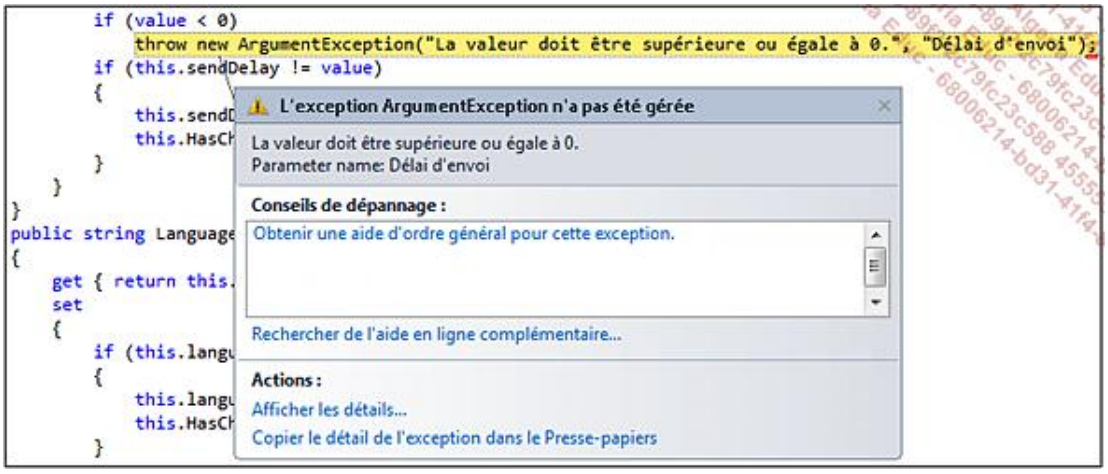
`ArgumentException` est un type fourni par le Framework .NET. Elle accepte en paramètres un message et le nom du paramètre qui est en cause.

Une exception est déclenchée avec le mot clé `throw` qui indique que l'exécution de la méthode en cours doit être arrêtée et l'exception transférée à la méthode appelante.

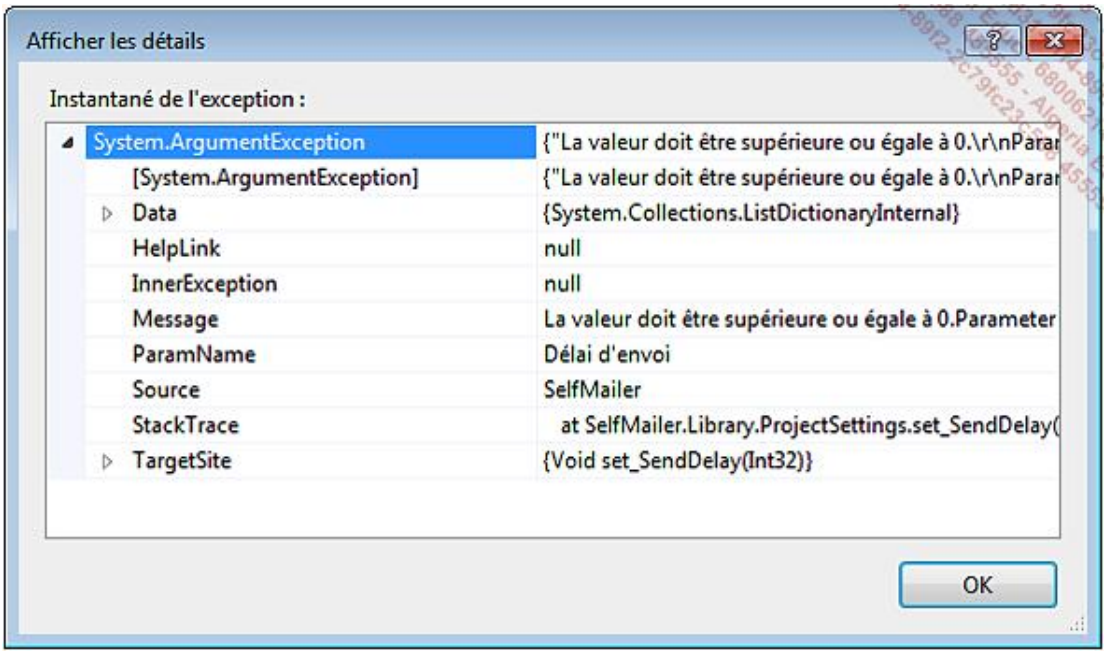
Dans notre application, cette spécificité est gérée de manière plus élégante au niveau du formulaire `ProjectSettings`. Le contrôle `SendDelay` de type `NumericUpDown` a sa propriété `Minimum` assignée à la valeur zéro. L'utilisateur ne recevra donc pas de message d'erreur. Insérer le code suivant au début de la méthode `Main` de la classe `Program` :

```
Project = new Library.Project();
Project.ProjectSettings.SendDelay = -1;
```

Lancez l'application (**F5**) pour déclencher l'erreur. Celle-ci n'étant pas gérée, cela entraîne une erreur fatale. Comme l'application est en mode débogage, l'application s'arrête sur la ligne qui déclenche l'exception avec les détails de celle-ci :



En cliquant sur le lien **Afficher les détails...**, une fenêtre s'ouvre permettant de consulter les valeurs de l'objet `ArgumentException` :



L’interception et la gestion des exceptions

Les exceptions font partie intégrante du cycle de vie d’une application. Si elles ne peuvent pas toujours être évitées, il est par contre indispensable de les intercepter et de les gérer d’une manière n’empêchant pas la suite de l’exécution de l’application. Le bloc `try ... catch ... finally` permet d’une part d’intercepter les erreurs et d’autre part de les traiter :

```
try
{
    Instructions.
}
catch
{
    Instructions de gestion d’une erreur.
}
finally
{
    Instructions toujours exécutées.
}
```

La partie `try` contient les instructions qui peuvent déclencher une erreur. Si une exception est déclenchée directement via une instruction `throw` ou depuis la pile des appels, l’exécution passe immédiatement dans la partie `catch` afin de traiter l’erreur. Qu’une exception soit levée ou non, la partie `finally` est toujours exécutée.

Les blocs `finally` contiennent les instructions qui seront exécutées en toutes circonstances. C’est dans ce bloc que les ressources doivent être libérées ou que les données doivent être enregistrées.

Lorsqu’une exception est déclenchée, une référence à l’objet `Exception` est disponible et peut être captée au niveau de la partie `catch` :

```
try
{
    Project = new Library.Project();
    Project.ProjectSettings.SendDelay = -1;
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Toutes les exceptions dérivant de `System.Exception`, le bloc `catch` de l’exemple précédent traitera toutes les erreurs. Il est possible de spécifier un type plus précis afin de gérer l’erreur d’une manière différente :

```
catch (ArgumentException ex)
{
    MessageBox.Show(ex.Message);
    Project.ProjectSettings.SendDelay = 0;
}
```

Un gestionnaire d’erreurs peut contenir plusieurs parties `catch`. Un seul d’entre eux sera exécuté par exception en fonction du type de l’exception :

```
try
{
    Project = new Library.Project();
    Project.ProjectSettings.SendDelay = -1;
}
catch (ArgumentException ex)
{
    MessageBox.Show(ex.Message);
    Project.ProjectSettings.SendDelay = 0;
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Dans cet exemple, si une erreur du type `ArgumentException` est déclenchée, le premier bloc `catch` sera exécuté car le type de l'exception géré correspond au type de l'exception déclenché. Si le type de l'erreur est différent, le second bloc sera exécuté.

Si aucun bloc `catch` ne correspond au type de l'exception levée, celle-ci est transmise suivant la pile des appels.

La disposition des blocs `catch` est importante car ceux-ci sont évalués dans le même ordre que leur ordre de déclaration. En inversant les deux blocs `catch` de l'exemple précédent, quelle que soit l'exception déclenchée, le bloc `catch` qui traite les types d'erreurs `Exception` sera exécuté même si le type réel de l'exception est `ArgumentException` car toute exception dérive à un niveau plus ou moins lointain de la classe de base `Exception`. C'est pourquoi vous devez déclarer les blocs `catch` du plus spécifique au plus générique et terminer par une gestion d'erreur générale.

Les blocs `catch` peuvent également être imbriqués.

Créez une nouvelle classe statique nommée `MailerTools` dans le dossier **Library** et insérez la méthode suivante :

```
public static void Send()
{
    try
    {
        foreach (MailProperties aMailProperties in
Program.Project.MailProperties)
        {
            if (aMailProperties.SendType != "Ne pas envoyer" &&
Program.Project.ProjectSettings.LanguageField != "[Aucune
langue]")
            {
                MailMessage MM = new MailMessage();
                MM.IsBodyHtml = true;
                SmtpClient SC = new SmtpClient();
                MM.From = new
MailAddress(Program.Project.MailServerSettings.FromEmail,
Program.Project.MailServerSettings.FromName);
                SC.Host =
Program.Project.MailServerSettings.Host;
                SC.Credentials = new
NetworkCredential(Program.Project.MailServerSettings.Username,
Program.Project.MailServerSettings.Password);

                foreach (DataRow aRow in
Program.Project.Data.Rows)
                {
                    try
                    {
                        string RowLang =
aRow[Program.Project.ProjectSettings.LanguageField].ToString();
                        if ((RowLang == aMailProperties.Name &&
aMailProperties.Name != "[Aucune langue]") ||
aMailProperties.Name == "[Aucune langue]")
                        {
                            MM.To.Clear();
                            MM.To.Add(new
MailAddress(aRow[Program.Project.ProjectSettings.EmailField].ToSt
ring()));

                            StringBuilder Body = new
StringBuilder();
Body.Append(aMailProperties.Body);
                            MM.Subject = aMailProperties.Subject;
                            MM.Body = Body.ToString();

                            if (MM.To.Count >= 1)
                            {
                                try
                                {
                                    //SC.Send(MM);

System.Threading.Thread.Sleep(Program.Project.ProjectSettings.Sen
dDelay);
                                }
                                catch (Exception ex)
```

```
                                {

                                    throw new
ProjectException(Program.Project , "Erreur lors de l'envoi", ex);
                                }
                            }
                        }
                    }
                    catch (ProjectException ex)
                    {
                        throw ex;
                    }
                    catch (Exception ex)
                    {
                        throw new
ProjectException(Program.Project , "Erreur lors de l'envoi", ex);
                    }
                }
            }
        }
    }
    catch (ProjectException ex)
    {
        throw ex;
    }
    catch (Exception ex)
    {
        throw new ProjectException(Program.Project , "Erreur lors
de l'envoi", ex);
    }
}
```

Dans cet exemple, si les blocs `try` imbriqués déclenchent une exception, elle sera gérée par leur propre bloc `catch`. Celui-ci lève une nouvelle exception de type `ProjectException` en spécifiant l'erreur originale dans le constructeur. Le bloc `try` suivant passe l'exécution à son bloc `catch` qui traite l'erreur de type `ProjectException` en la redéclenchant. L'erreur d'origine est alors accessible via la propriété `InnerException`.

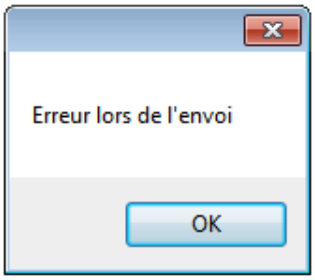
Ajoutez le code suivant au constructeur du formulaire **Send** pour gérer l'erreur :

```
try
{
    Library.MailerTools.Send();
}
catch (Exception ex)
{
    while (ex != null)
    {
        MessageBox.Show(ex.Message);
        ex = ex.InnerException;
    }
}
```

Ajoutez également le code d'ouverture du formulaire dans la méthode `envoyerToolStripMenuItem_Click` du formulaire **Main** :

```
new Forms.Send().ShowDialog();
```

Lancez l'application (**F5**) et cliquez sur le menu **Envoyer** pour afficher les messages d'erreurs :



Le traçage

Le traçage d'application permet d'enregistrer des informations sur l'état de l'application sans l'interrompre. Les applications complexes ne peuvent pas forcément être examinées ligne par ligne pour identifier une erreur de logique. Les classes `Debug` et `Trace` de l'espace de noms `System.Diagnostics` exposent des méthodes statiques permettant de tester les conditions d'exécutions et d'enregistrer des messages qui seront affichés dans la fenêtre **Sortie** du débogueur. Ces informations sont également transmises à leur collection d'écouteurs partagée entre les deux classes.

1. Les classes Debug et Trace

Les classes `Debug` et `Trace` sont identiques fonctionnellement. La différence principale est que les instructions de la classe `Debug` ne sont pas incluses par défaut lors de la compilation en mode **release** contrairement à celles de la classe `Trace`. La classe `Debug` sera donc préférée pour le débogage de l'application et la classe `Trace` sera utilisée pour le suivi et l'optimisation après la finalisation de l'application.

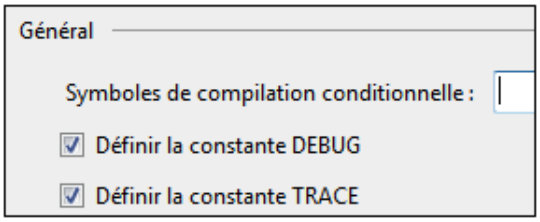
Les méthodes de la classe `Debug` sont marquées avec l'attribut `Conditional` précisant la constante `DEBUG` permettant de spécifier que la constante doit être définie pour compiler l'instruction :

```
[Conditional("DEBUG")]
```

Les méthodes de la classe `Trace` sont également marquées avec l'attribut `Conditional` mais la constante est `TRACE` :

```
[Conditional("TRACE")]
```

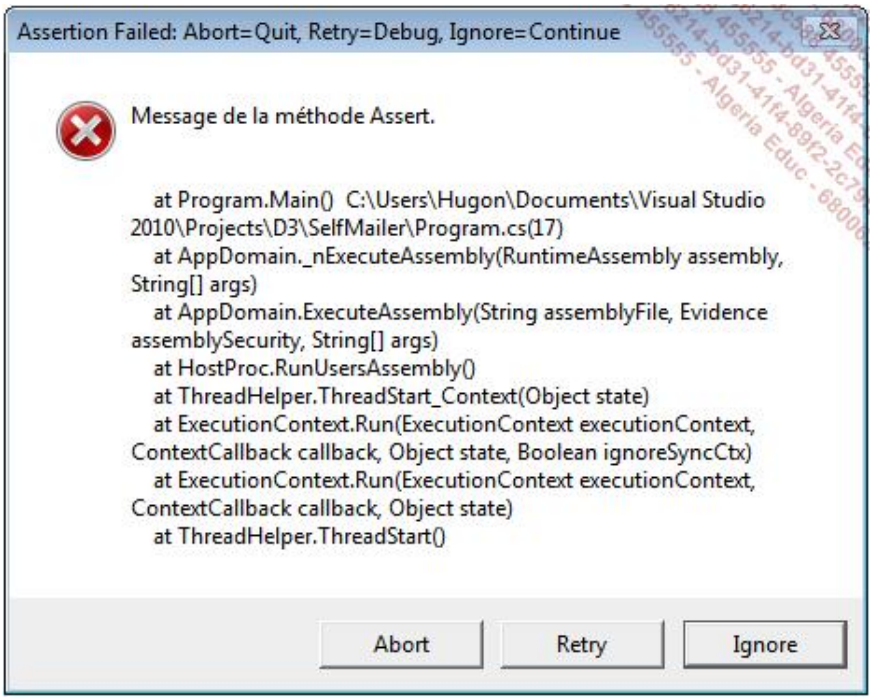
Ces constantes sont définies dans les propriétés du projet sous l'onglet **Générer**. Par défaut, en mode de compilation **debug**, les deux constantes `DEBUG` et `TRACE` sont définies alors que pour le mode de compilation **release**, seule la constante `TRACE` est définie par défaut :



Les deux classes exposent une série de méthodes pour écrire dans la collection d'écouteurs :

- `Assert` : vérifie une condition. Si elle est fausse, un message est inscrit dans la collection d'écouteurs et une boîte de dialogue est affichée.

```
Trace.Assert(false, "Message de la méthode Assert.");
```



- `Fail` : comme pour la méthode `Assert`, un message est inscrit dans la collection d'écouteurs et une boîte de dialogue est affichée mais sans vérification de condition.

```
Trace.Fail("Message de la méthode Fail.");
```

- `Write` : écrit un message dans la collection d'écouteurs.

```
Trace.Write("Message de la méthode Write.");
```

- `WriteIf` : écrit un message dans la collection d'écouteurs si la condition est vraie.

```
Trace.WriteIf(true, "Message de la méthode WriteIf.");
```

- `WriteLine` et `Print` : écrivent un message avec un terminateur de ligne dans la collection d'écouteurs.

```
Trace.WriteLine(true, "Message de la méthode WriteLine.");
```

- `WriteLineIf` : écrit un message avec un terminateur de ligne dans la collection d'écouteurs si la condition est vraie.

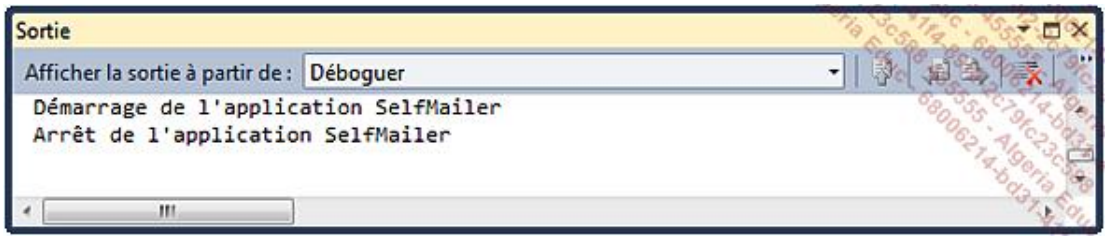
```
Trace.WriteLineIf(true, "Message de la méthode WriteLineIf.");
```

Les classes `Debug` et `Trace` disposent également de méthodes `Indent` et `Unindent` permettant de modifier l'indentation des messages qui seront écrits d'une unité. Cette unité d'indentation est définie dans la propriété `IndentSize` et le niveau total d'indentation peut être défini directement avec la propriété `IndentLevel`.

Ajoutez les instructions suivantes au début et à la fin de la méthode `Main` de la classe `Program` :

```
Trace.WriteLine("Démarrage de l'application SelfMailer");
...
Trace.WriteLine("Arrêt de l'application SelfMailer");
```

Lancez l'application (**F5**) et examinez la fenêtre **Sortie** (menu **Déboguer** - **Fenêtres** - **Sortie**) :



2. La collection d'écouteurs

Tous les messages inscrits par les méthodes des classes `Debug` et `Trace` sont écrits dans la collection d'écouteurs (propriété `Listeners`) de type `TraceListenerCollection`.

La collection d'écouteurs est instanciée avec un membre par défaut de type `DefaultTraceListener` permettant ainsi de recevoir les messages de traçage sans paramétrage supplémentaire. Cet écouteur par défaut s'occupe de transmettre les messages au débogueur de Visual Studio et une fois que l'application se termine, les messages enregistrés par cet écouteur sont supprimés et il devient impossible de les consulter plus tard.

a. La création d'écouteurs

Pour sauvegarder le traçage d'une application, le Framework .NET met à disposition des classes spécialisées permettant d'écrire la sortie aux formats texte avec la classe `TextWriterTraceListener`, XML avec la classe `XmlWriterTraceListener` ou dans les journaux d'évènements de Windows avec la classe `EventLogTraceListener`. Il suffit d'instancier un objet du type souhaité puis de l'ajouter à la collection d'écouteurs :

```
TextWriterTraceListener textListener =
                new TextWriterTraceListener(@"C:\Trace.txt");
Trace.Listeners.Add(textListener);

XmlWriterTraceListener xmlListener =
                new XmlWriterTraceListener(@"C:\Trace.xml");
Trace.Listeners.Add(xmlListener);

EventLogTraceListener logListener =
                new EventLogTraceListener("Application");
Trace.Listeners.Add(logListener);
```

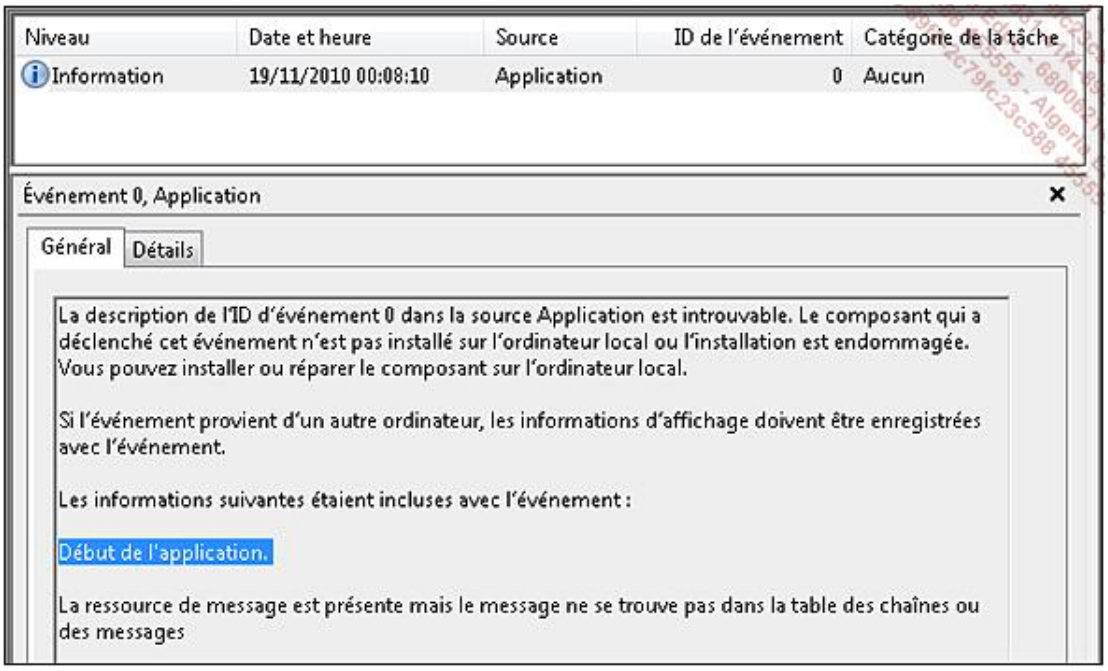
b. La sauvegarde des traces

Après avoir inscrit des messages de traçage, la méthode `Flush` doit être appelée sur la classe `Trace` ou `Debug` pour lancer l'écriture sur fichier ou dans les journaux d'évènements :

```
Trace.Write("Début de l'application.");
Trace.Flush();
```

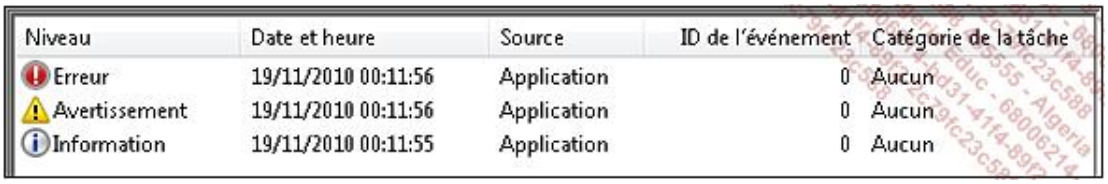
La propriété booléenne `AutoFlush` permet d'indiquer si le vidage de la mémoire tampon et l'écriture sur fichiers ou dans les journaux d'évènements se font automatiquement ou non.

L'exemple précédent produira un fichier texte indiquant le message, un fichier XML plus complexe indiquant des informations supplémentaires et un évènement dans le journal **Application** de Windows :

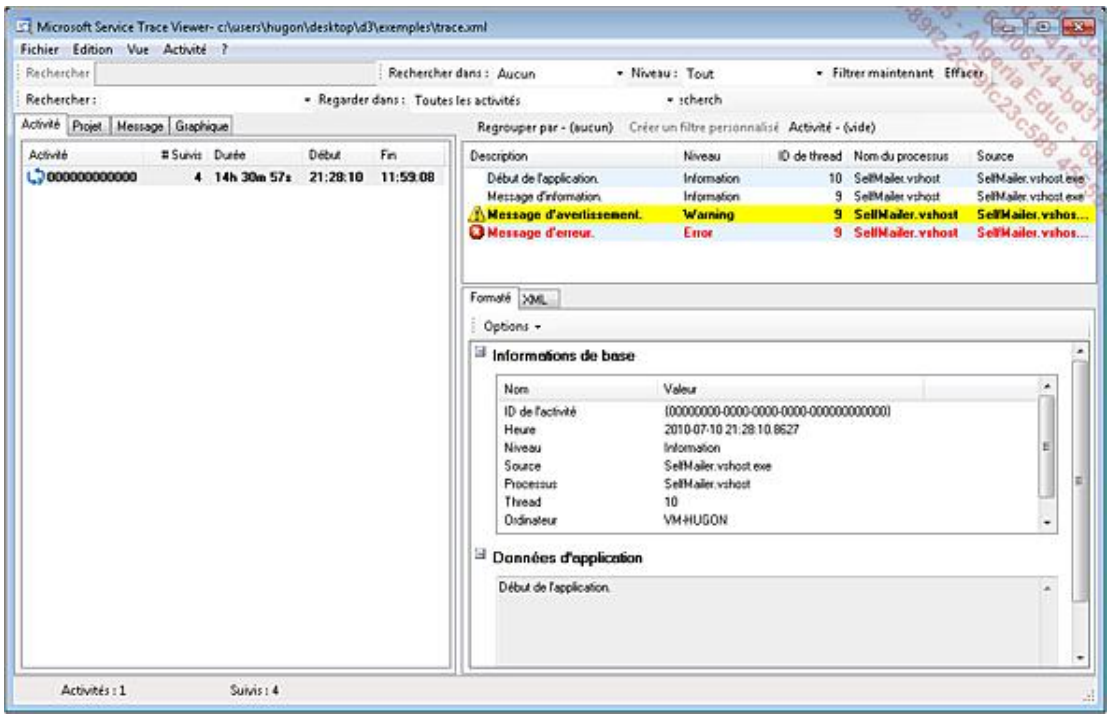


La classe `Trace` contient trois méthodes supplémentaires permettant de définir le type du message : information, avertissement ou erreur :

```
Trace.TraceInformation("Message d'information.");
Trace.TraceWarning("Message d'avertissement.");
Trace.TraceError("Message d'erreur.");
```



Outre les fichiers texte et les journaux d'évènements Windows, Visual Studio met à disposition l'outil **Service Trace Viewer** (**Tous les programmes** - **Microsoft Visual Studio 2010** - **Microsoft Windows SDK Tools**). Cet outil permet d'analyser en particulier les traces WCF mais aussi les traces classiques au format XML :



3. Les commutateurs de trace

En phase de débogage, il est intéressant de disposer de toutes les informations de traçage mais une fois l'application déployée, le traçage ne devient utile que pour les cas particuliers. Les commutateurs de trace permettent de configurer l'inscription ou non des informations de traçage.

a. Le fonctionnement des commutateurs de trace

Il existe deux principaux commutateurs de trace : le commutateur de type `BooleanSwitch` qui permet de spécifier un état activé ou non et le type `TraceSwitch` qui permet un réglage plus précis sur cinq niveaux. Les constructeurs des commutateurs reçoivent deux paramètres : leur nom affiché (`DisplayName`) et leur description (`Description`) :

```
BooleanSwitch booleanSwitch=new
BooleanSwitch("BooleanSwitch","Commutateur booléen.");

TraceSwitch traceSwitch=new
TraceSwitch("TraceSwitch","Commutateur complexe.");
```

Les commutateurs de trace peuvent ensuite être utilisés en tant que conditions pour l'écriture de messages. La liaison n'est pas automatique, il faut donc utiliser les méthodes conditionnelles de la classe `Trace` :

```
Trace.WriteLineIf(booleanSwitch.Enabled, "Démarrage de
l'application SelfMailer");
```

La classe `TraceSwitch` compte cinq niveaux de trace, définis de 0 à 4 dans la propriété `Level` de type `TraceLevel` :

- `Off` : correspond à la valeur 0. Aucun message n'est envoyé en sortie.
- `Error` : correspond à la valeur 1. Les messages d'erreur sont envoyés à la sortie.
- `Warning` : correspond à la valeur 2. Les messages d'avertissement sont envoyés à la sortie.
- `Info` : correspond à la valeur 3. Les messages d'information sont envoyés à la sortie.
- `Verbose` : correspond à la valeur 4. Tous les messages sont envoyés à la sortie.

Lorsque vous souhaitez déterminer si un message doit être écrit dans la sortie, notamment avec la méthode `WriteLineIf` de la classe `Trace`, vous pouvez soit vérifier la valeur de la propriété `Level` de la classe `TraceSwitch` soit les propriétés booléennes exposées : `TraceError`, `TraceWarning`, `TraceInfo` ou `TraceVerbose`. Les deux instructions suivantes sont équivalentes :

```
Trace.WriteLineIf(traceSwitch.TraceInfo, "Arrêt de l'application
SelfMailer");
```

```
Trace.WriteLineIf(traceSwitch.Level == TraceLevel.Info, "Arrêt de
l'application SelfMailer");
```

Dès qu'un niveau de traçage est défini, les propriétés booléennes de la classe `TraceSwitch` sont mises à jour et tous les niveaux inférieurs prennent la valeur `true`.

b. La configuration des commutateurs de trace

La déclaration des commutateurs de trace est effectuée dans le code de l'application mais leur configuration doit se faire en dehors du code de manière à pouvoir les activer après la compilation de l'application.

La configuration des commutateurs de trace se fait dans le fichier de configuration de l'application. Ce fichier, au format XML, permet de paramétrer l'application. Il est nommé **App.config** dans le projet, puis une fois compilé, ce fichier porte le nom **[nom de l'exécutable de l'application].exe.config**.

Ajoutez un fichier de configuration au projet **SelfMailer** : menu **Projet** - **Ajouter un nouvel élément...** puis sélectionnez **Fichier de configuration de l'application** dans la fenêtre **Ajouter un nouvel élément** et cliquez sur le bouton **Ajouter**.

Lorsque le code crée un commutateur, le fichier de configuration est analysé. La valeur de la propriété `DisplayName` permet de faire la liaison entre un commutateur déclaré dans le code et sa configuration dans le fichier XML. Ajoutez le code suivant au fichier de configuration :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="BooleanSwitch" value="1"/>
      <add name="TraceSwitch" value="3"/>
    </switches>
  </system.diagnostics>
</configuration>
```

Pour les objets de type `BooleanSwitch`, la valeur 0 indique que le commutateur est désactivé et toute autre valeur non nulle représente l'état actif.

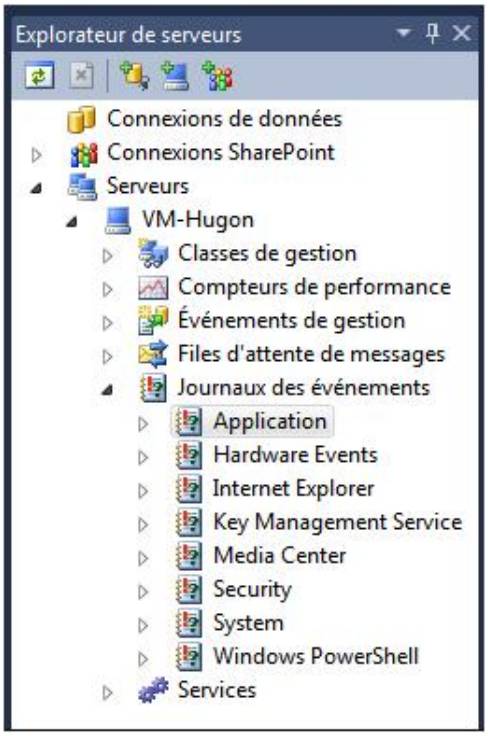
Pour les objets de type `TraceSwitch`, les valeurs vont de 0 à 4 inclus pour déterminer le niveau de trace souhaité. Toute valeur supérieure à 4 est traitée comme la valeur 4 (`TraceLevel.Verbose`).

Les journaux d'évènements

Les journaux d'évènements représentent le point central des messages permettant de vérifier l'état de santé du système et des applications. Les erreurs d'une application doivent inscrire des messages dans les journaux d'évènements dans un but de centralisation et de facilité de suivi du cycle de vie de l'application.

1. L'interaction avec les journaux d'évènements

Les journaux d'évènements sont accessibles depuis Visual Studio dans l'**Explorateur de serveurs** (**Ctrl** + **W**, **L**) :



Un évènement est notamment défini par les propriétés suivantes :

- Category : une catégorie peut être définie pour faciliter le filtrage des évènements.
- Date : correspond à la date d'entrée dans le journal d'évènements.
- EntryType : indique le type du message : Information, Warning ou Error. Deux types supplémentaires sont utilisés uniquement dans le journal de sécurité : FailureAudit et SuccesAudit.
- EventID : l'identifiant de l'évènement.
- Message : le message de l'évènement.
- Source : indique le nom du programme qui a inscrit l'évènement.

Dans la section précédente, nous avons vu comment écrire des messages dans un journal d'évènements grâce à la classe `EventLogTraceListener`. Cette classe contient un membre de type `EventLog` qui se charge de l'écriture dans le journal d'évènements. Ce type peut également être utilisé directement pour lire et écrire dans les journaux.

La classe `EventLog` fait partie de l'espace de noms `System.Diagnostics`. Elle permet de lire et écrire dans les journaux d'évènements. Une entrée est représentée par la classe `EventLogEntry` disponible au travers de la propriété `Entries` de type `EventLogEntryCollection` de la classe `EventLog`.

2. La gestion des journaux d'évènements

La classe `EventLog` permet de gérer les journaux d'évènements grâce aux méthodes statiques suivantes :

- `CreateEventSource` : permet de créer une nouvelle source d'évènements ainsi qu'un nouveau journal.

```
CreateEventSource("SelfMailer", "Mes applications");
```

- `Delete` : supprime complètement un journal.

```
EventLog.Delete("Mes applications");
```

- `DeleteEventSource` : supprime une source d'évènements.

```
EventLog.DeleteEventSource("SelfMailer");
```

- `Exists` : détermine si le journal existe sur la machine locale.

```
EventLog.Exists("Mes applications");
```

- `GetEventLogs` : retourne tous les journaux d'évènements.

```
EventLog.GetEventLogs();
```

- `LogNameFromSourceName` : retourne le nom du journal associé à la source spécifiée.

```
EventLog.LogNameFromSourceName("SelfMailer", ".");
```

- `SourceExists` : détermine si la source existe sur la machine locale.

```
EventLog.SourceExists("SelfMailer");
```

La première chose à faire avant d'écrire des évènements est de créer une source en vérifiant que celle-ci n'existe pas déjà. La source sera utilisée par l'application pour identifier facilement les entrées du journal. L'exemple suivant crée la source **SelfMailer** pour le journal **Application** si elle n'existe pas :

```
if (!EventLog.SourceExists("SelfMailer"))
{
    EventLog.CreateEventSource("SelfMailer", "Mes applications");
}
```

La création de sources demande des droits d'administrateur sur la machine, c'est pourquoi il est préférable de la créer pendant le déploiement de l'application.

3. L'écriture d'évènements

Pour écrire de nouvelles entrées dans un journal d'évènements, la classe `EventLog` expose la méthode statique et d'instance `WriteEntry`. La version statique de la méthode `WriteEntry` prend en paramètre la source qui est donnée lors de l'instanciation pour la version instanciée de la méthode `WriteEntry` :

```
EventLog eventLog = new EventLog("Mes applications",
                                 ".",
                                 "SelfMailer");
```

La méthode `WriteEntry` comporte de nombreuses surcharges permettant d'écrire une nouvelle entrée avec plus ou moins de spécifications :

- Écriture d'un message informatif :

```
eventLog.WriteEntry("Mon message");
```

- Écriture d'un message avec précision du type :

```
eventLog.WriteEntry("Mon message", EventLogEntryType.Warning);
```

- Écriture d'un message avec précision du type et de l'identificateur (0 par défaut) :

```
eventLog.WriteEntry("Mon message", EventLogEntryType.Warning, 3);
```

- Écriture d'un message avec précision du type, de l'identificateur (0 par défaut) et de la catégorie (0 par défaut) :

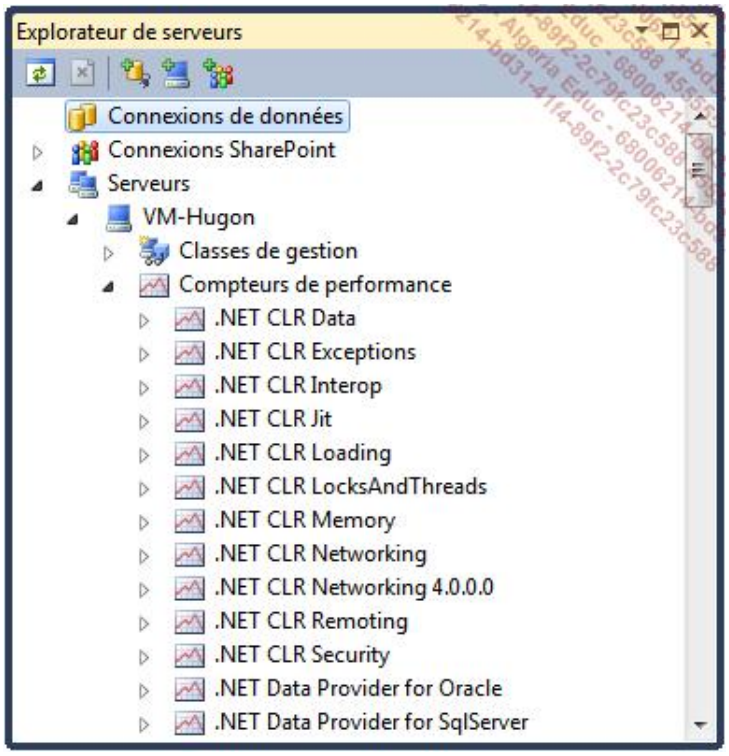
```
eventLog.WriteEntry("Mon message",
                    EventLogEntryType.Warning, 3, 5);
```

Une dernière surcharge permet également de passer un tableau de `byte` contenant des informations supplémentaires.

Les compteurs de performance

Surveiller une application pendant son cycle de vie est très important pour apporter des améliorations sur la performance, augmenter la rapidité et optimiser l'espace mémoire utilisé. Les compteurs de performance sont indispensables pour vérifier et contrôler ces points.

Les compteurs de performance sont accessibles à partir du menu **Affichage** - **Explorateurs de serveurs** (**Ctrl** + **W**, **L**) :



L'espace de noms `System.Diagnostics` fournit des classes permettant d'interagir avec les compteurs de performance. Parmi les plus utilisées :

- `PerformanceCounter` : cette classe peut être utilisée à la fois pour surveiller et pour modifier les compteurs.
- `PerformanceCounterCategory` : cette classe permet de gérer et parcourir les catégories de compteurs.

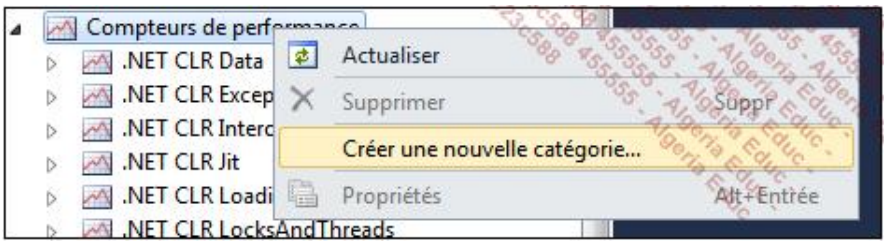
Vous pouvez utiliser ces compteurs dans votre application ou en créer de nouveaux spécifiques à une application pour pouvoir les analyser.

1. La création de compteurs de performance

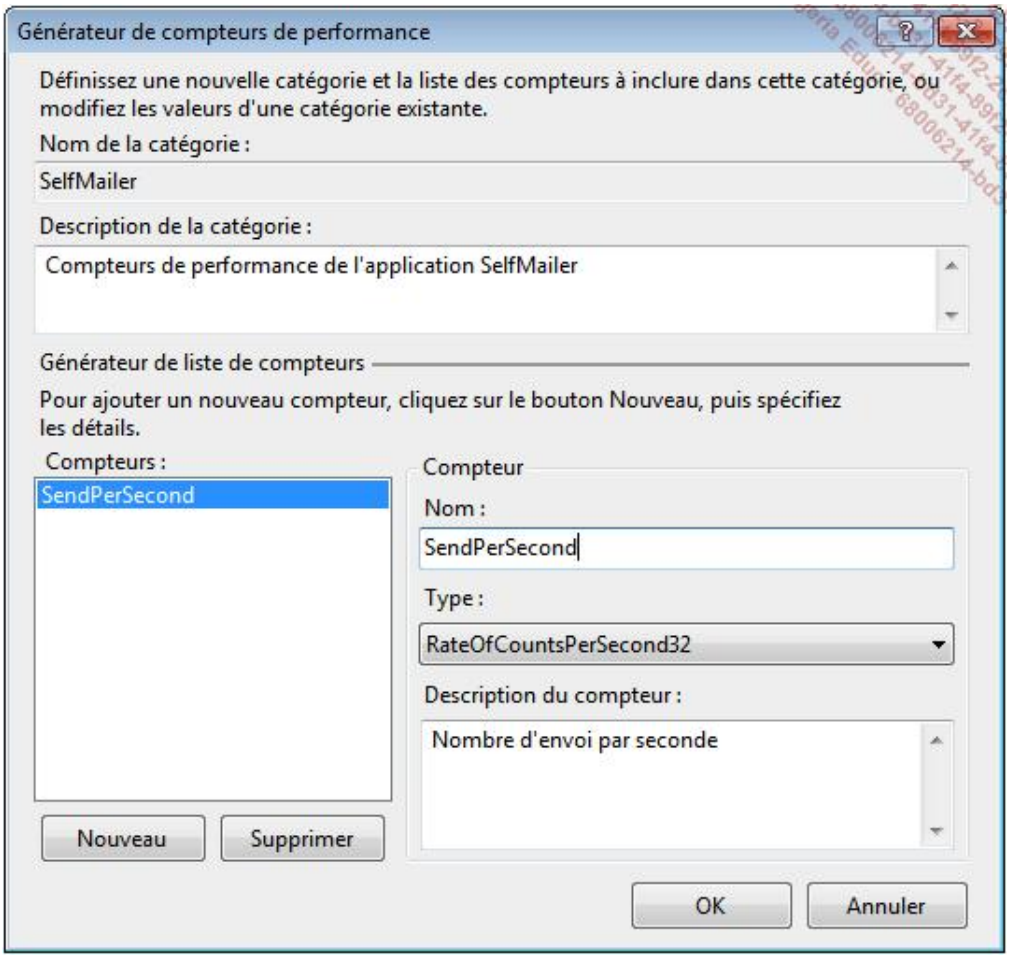
a. Depuis Visual Studio

Les compteurs de performance peuvent être créés directement depuis Visual Studio, dans la fenêtre **Explorateurs de serveurs** (**Ctrl** + **W**, **L**).

Ouvrez le menu contextuel du nœud **Compteurs de performance** puis cliquez sur le menu **Créer une nouvelle catégorie...** :



La fenêtre **Générateur de compteurs de performance** s'ouvre. Spécifiez **SelfMailer** pour le nom de la catégorie et ajoutez une description à la catégorie. Pour pouvoir terminer la création de la catégorie, il faut créer au moins un compteur. Cliquez sur le bouton **Nouveau** et saisissez **SendPerSecond** pour le nom du compteur. Sélectionnez **RateOfCountsPerSecond32** pour le champ **Type** et donnez une description à ce compteur :



Après validation du formulaire, la catégorie et le compteur apparaissent dans la liste des compteurs de performance :

b. Depuis le code

La création de compteurs de performance peut également se faire dynamiquement depuis le code. Voici la méthode permettant de créer le même compteur **SelfMailer** que précédemment depuis l'interface Visual Studio :

```
public void CreateCounter()
{
    if (PerformanceCounterCategory.Exists("SelfMailer"))
    {
        PerformanceCounterCategory.Delete("SelfMailer");
    }

    CounterCreationData counter =
        new CounterCreationData("SendPerSecond",
            "Nombre d'envoi par seconde",
            PerformanceCounterType.RateOfCountsPerSecond32);

    CounterCreationDataCollection counters =
        new CounterCreationDataCollection();
    counters.Add(counter);
```

```
    PerformanceCounterCategory.Create("SelfMailer",
        "Compteurs de performance de l'application SelfMailer",
        PerformanceCounterCategoryType.SingleInstance,
        counters);
}
```

Si vous tentez de créer une catégorie déjà existante, une exception du type `InvalidOperationException` sera levée. C'est pourquoi la méthode statique `Exists` de la classe `PerformanceCounterCategory` doit être utilisée de manière à vérifier que le nom de la catégorie n'est pas déjà utilisé. De la même manière, il n'est pas permis de créer deux compteurs avec le même nom pour une catégorie, cela lèverait une exception de type `ArgumentException`.

La création d'un compteur se fait en instanciant un objet du type `CounterCreationData`. Cet objet est ensuite ajouté à une collection d'objets de type `CounterCreationDataCollection`. La méthode statique `Create` de la classe `PerformanceCounterCategory` permet de spécifier le nom de la catégorie, sa description, son type ainsi que la collection de compteurs créée précédemment.

L'énumération `PerformanceCounterCategoryType` permet d'indiquer si la catégorie peut avoir plusieurs instances. Elle possède trois valeurs :

- `Unknown` : spécifie que les fonctionnalités d'instances sont inconnues.
- `SingleInstance` : spécifie que la catégorie n'a qu'une seule instance, c'est-à-dire que les compteurs sont globaux.
- `MultiInstance` : spécifie que la catégorie peut avoir plusieurs instances et que les compteurs ne sont pas globaux.

2. L'utilisation de compteurs de performance

Les compteurs de performance peuvent être ajoutés à un formulaire Windows en faisant un glissé-déposé depuis la fenêtre **Boîte à outils** d'un objet **PerformanceCounter** (sous l'onglet **Composants**) sur le concepteur de vue de Visual Studio. L'instance d'objet du type `PerformanceCounter` apparaît dans la barre des composants du concepteur de vue et la fenêtre **Propriétés** permet de le configurer. Il est également possible de faire un glissé-déposé depuis la fenêtre **Explorateur de serveurs** vers le concepteur de vue en sélectionnant le compteur souhaité. Dans ce cas, les propriétés de l'objet comme le nom de la catégorie, le nom du compteur et le nom de la machine sont déjà remplies.

Ajoutez un compteur de performance sur le formulaire **Send** et ayant les propriétés suivantes :

Propriété	Valeur
`Name`	SendPerSecondCounter
`CategoryName`	SelfMailer
`CounterName`	SendPerSecond
`MachineName`	.

Le caractère '.' pour spécifier une machine indique qu'il s'agit de la machine locale.

Ajoutez également un contrôle **Label** avec les propriétés suivantes :

Propriété	Valeur
`Name`	SendPerSecondValue
`Text`	0 email / seconde

Ajoutez un composant **Timer** permettant de mettre à jour le texte du **Label SendPerSecondValue** :

Propriété	Valeur

Name	SendPerSecondTimer
Interval	1000
Enabled	True

Ajoutez un gestionnaire pour l'évènement `Tick` du **Timer SendPerSecondTimer** permettant de mettre à jour la valeur du **Label SendPerSecondValue** :

```
private void SendPerSecondTimer_Tick(object sender, EventArgs e)
{
    this.SendPerSecondValue.Text =
          this.SendPerSecondCounter.NextValue().ToString()
          + " email / seconde";
}
```

Le code du formulaire **Send** utilise un objet `BackgroundWorker` pour exécuter la méthode d'envoi dans un autre thread et permettre la mise à jour du formulaire **Send**. Ce type de composant sera étudié plus loin dans l'ouvrage.

Maintenant que nous pouvons lire la valeur d'un compteur et l'afficher sur un formulaire, il faut que ce compteur soit incrémenté. Ajoutez un objet statique de type `PerformanceCounter` dans la classe `MailerTools` :

```
private static PerformanceCounter SendPerSecondCounter =
    new PerformanceCounter("SelfMailer", "SendPerSecond", false);
```

La classe `PerformanceCounter` expose les méthodes `Increment`, `IncrementBy` et la propriété `RawValue` pour gérer la valeur du compteur :

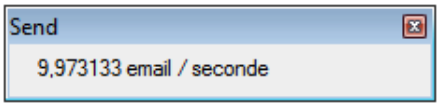
- `Increment` : méthode permettant d'incrémenter de 1 la valeur du compteur.
- `IncrementBy` : méthode permettant d'incrémenter la valeur du compteur du nombre spécifié en paramètre.
- `RawValue` : propriété permettant de lire et d'assigner une valeur au compteur.

Pour modifier la valeur d'un compteur, sa propriété `ReadOnly` doit avoir la valeur `False`.

Ajoutez un incrément lors de l'envoi d'un email dans la méthode `Send` de la classe `MailerTools` :

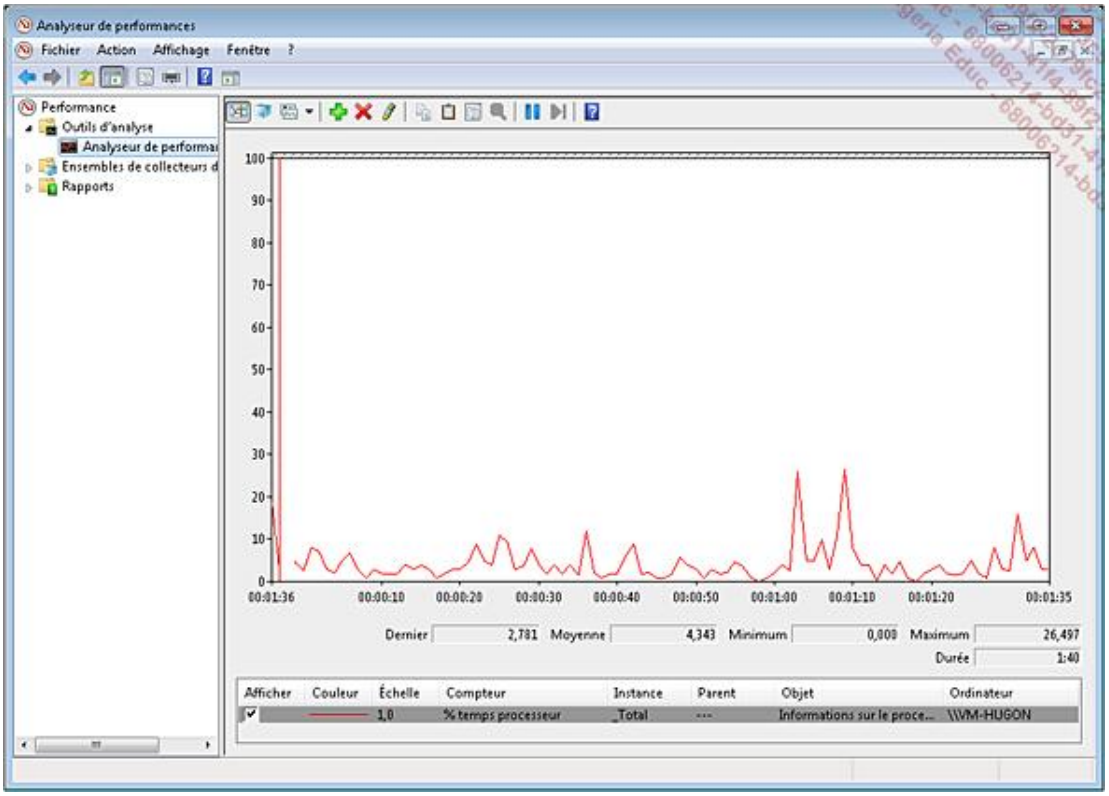
```
...
try
{
    //SC.Send(MM);
    SendPerSecondCounter.Increment();
    Thread.Sleep(Program.Project.ProjectSettings.SendDelay);
}
...
```

Lancez l'application (**F5**). Après la configuration des différents paramètres et l'insertion d'un jeu de données, cliquez sur le menu **Envoyer**. La procédure d'envoi se lance, le compteur est incrémenté et le formulaire **Send** met à jour la valeur du taux d'envoi par seconde :



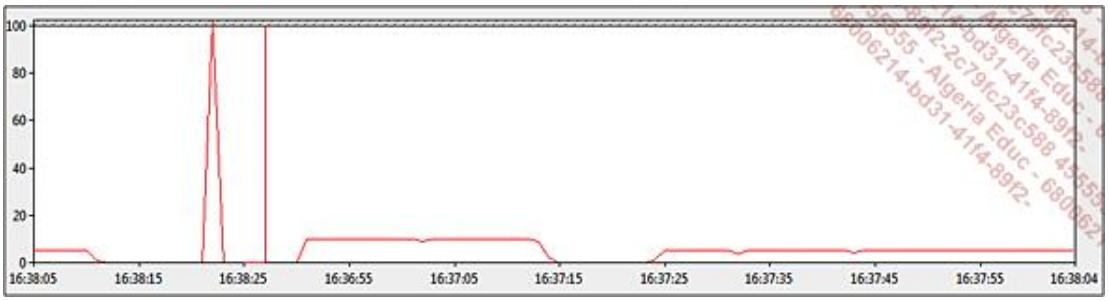
3. L'analyse de compteurs de performance

Windows contient un outil permettant de suivre les compteurs de performance. Lancez le **Moniteur de performance** depuis le **Panneau de contrôle** - **Système et sécurité** - **Outils d'administration** :



Par défaut le compteur **% Processor Time** est chargé et affiche le pourcentage d'utilisation du processeur. Pour ajouter un compteur, cliquez sur le bouton **+** vert dans la barre d'outils en haut. Ouvrez la catégorie **SelfMailer**, sélectionnez le compteur **SendPerSecond** et cliquez sur le bouton **Ajouter**. Le compteur est ajouté dans la liste de droite de la fenêtre **Ajout de compteurs**. Cliquez sur le bouton **OK** pour valider l'ajout.

Désormais le compteur est affiché sous forme de graphique dans le moniteur de performance. Lancez l'application (**F5**) et après paramétrages, effectuez plusieurs envois consécutifs en faisant varier la valeur du délai d'envoi pour voir évoluer le graphique :



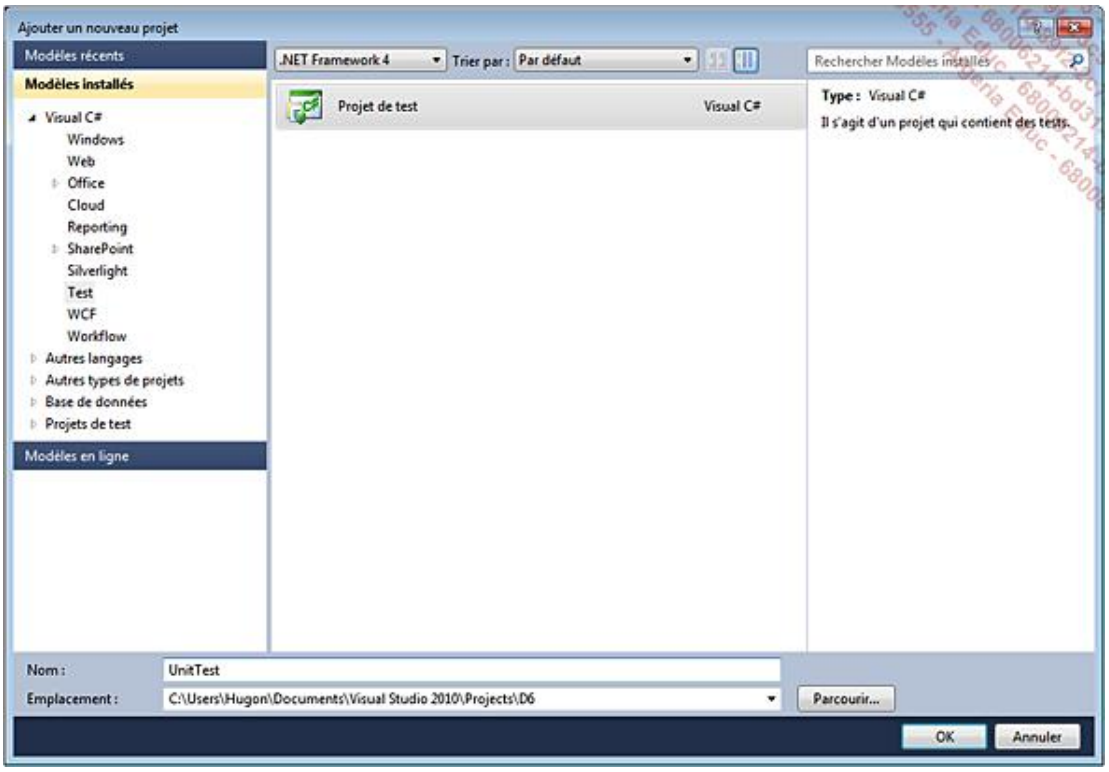
Introduction aux tests unitaires

Les tests unitaires représentent le meilleur moyen de tester le code d'une application tout au long de son développement. Ces tests permettent, d'une part, de s'assurer que les méthodes retournent les bons résultats en rapport avec les paramètres qui lui sont passés et, d'autre part, ils peuvent être utilisés pour faire du **Test Driven Development**. Il s'agit d'une technique où les tests sont écrits avant les classes et les méthodes. Ainsi, ce sont les tests qui vont définir le développement et plus l'inverse.

1. La création du projet

Visual Studio propose un modèle de projet pour les tests unitaires : **Projet de test**.

Ajoutez un projet de tests unitaires à la solution **SelfMailer** (menu **Fichier** - **Ajouter** - **Nouveau projet...** - **Test**) et nommez-le **UnitTest** :



Visual Studio crée la solution en ajoutant la référence à la librairie **Microsoft.VisualStudio.QualityTools.UnitTestFramework**, qui contient les classes nous permettant de créer les tests. Le fichier **UnitTest1.cs** est également ajouté. C'est dans ce fichier que seront écrits les tests.

Un nouveau dossier, **Solution Items**, apparaît désormais dans la solution, il contient trois fichiers : **Local.testsettings**, **SelfMailer.vsmdi**, **TraceAndTestImpact.testsettings**.

2. Les classes de tests unitaires

L'espace de noms utilisé pour les tests unitaires est `Microsoft.VisualStudio.TestTools.UnitTesting`. Il contient la principale classe permettant d'effectuer des tests : la classe statique `Assert` permettant de réaliser des assertions, c'est-à-dire donner une proposition vraie sur la méthode et s'assurer que le résultat est également correct. La classe `Assert` contient de nombreuses méthodes statiques pour créer les assertions :

- `AreEqual` : vérifie que deux valeurs sont identiques.
- `AreNotEqual` : vérifie que deux valeurs sont différentes.
- `AreNotSame` : vérifie que deux objets sont différents.

- `AreSame` : vérifie que deux objets sont identiques.
- `Fail` : fait échouer une assertion sans tester de conditions.
- `Inconclusive` : indique que le résultat de l'assertion n'est pas déterminant. Indique également que l'assertion n'est pas encore implémentée.
- `IsFalse` : vérifie qu'une expression est fausse.
- `IsInstanceOfType` : vérifie qu'un objet est une instance du type spécifié.
- `IsNotInstanceOfType` : vérifie qu'un objet n'est pas une instance du type spécifié.
- `IsNotNull` : vérifie qu'un objet n'est pas `null`.
- `IsNull` : vérifie qu'un objet est `null`.
- `IsTrue` : vérifie qu'une expression est vraie.

L'espace de noms expose également de nombreux attributs avec parmi eux, ceux qui seront utilisés dans la plupart des tests :

- `TestClassAttribute` : cet attribut doit être déclaré sur une classe et permet d'identifier celle-ci comme étant une classe de tests. Toutes les classes qui possèderont cet attribut seront évaluées lors de l'exécution des tests.
- `TestMethodAttribute` : cet attribut doit être déclaré sur une méthode et permet d'identifier celle-ci comme étant une méthode de tests. Toutes les méthodes qui possèderont cet attribut seront évaluées lors de l'exécution des tests.
- `TestInitializeAttribute` : cet attribut est déclaré sur une méthode permettant d'initialiser l'environnement de tests. Les méthodes marquées avec cet attribut seront exécutées avant chaque méthode de tests.
- `TestCleanupAttribute` : cet attribut est déclaré sur une méthode permettant de restaurer l'environnement de tests. Les méthodes marquées avec cet attribut seront exécutées après chaque méthode de tests.

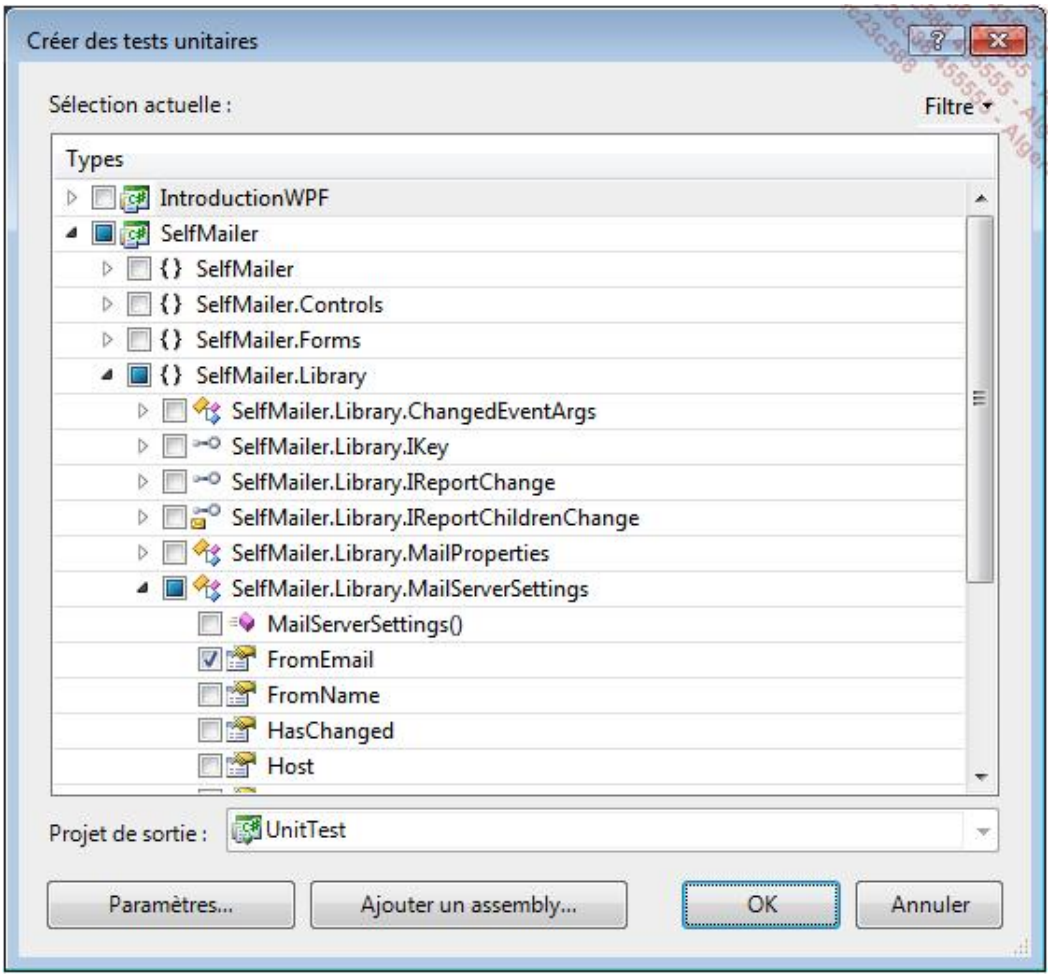
La classe `TestContext` va permettre de récupérer le contexte dans lequel le test se déroule et notamment des informations concernant le nom du test, le chemin des répertoires de log, le chemin d'exécution, etc. Pour récupérer une instance du type `TestContext`, il suffit de déclarer une variable de ce type dans la classe de test et elle sera automatiquement instanciée.

La mise en place d'une série de tests

1. L'ajout de tests au projet

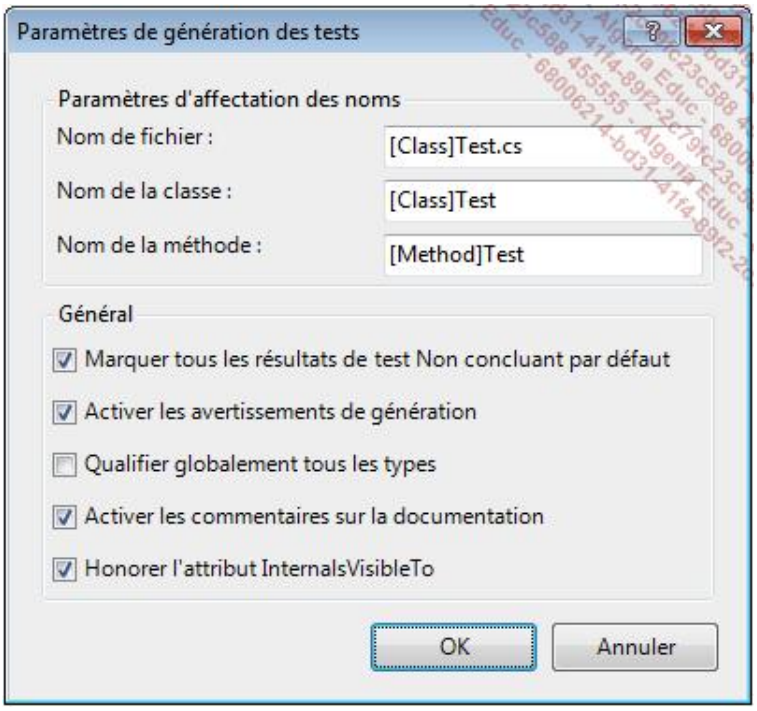
Le nombre de méthodes à tester dans un projet pouvant être très important, Visual Studio met à disposition l'assistant de création de tests, un outil permettant de gagner du temps dans l'écriture des méthodes de tests.

Ouvrez le menu contextuel du projet **UnitTest** puis sélectionnez le menu **Ajouter** - **Test unitaire....** La fenêtre **Créer des tests** unitaires s'ouvre et permet de choisir les méthodes à tester parmi les projets de la solution :



Cette fenêtre permet de sélectionner les méthodes qui devront être testées. Le champ **Projet de sortie** propose le choix parmi les projets de tests de la solution dans lequel seront créées les méthodes. Si la solution ne contient aucun projet de tests, l'assistant vous proposera d'en créer un nouveau.

Le bouton **Paramètres...** ouvre la fenêtre de paramétrage de la génération des classes et méthodes de tests. Vous pouvez notamment spécifier les noms des fichiers de sortie, des classes et des méthodes :



Il est possible de charger un assemblage qui n'est pas dans la solution en cliquant sur le bouton **Ajouter un assembly...**.

Sélectionnez la propriété `FromEmail` de la classe `MailServerSettings` de l'espace de noms `SelfMailer.Library` dans l'assistant de création de tests unitaires puis cliquez sur le bouton **OK**. Visual Studio génère la classe de test dans un fichier **MailServerSettingsTest.cs** et ajoute les références aux bibliothèques dont les tests auront besoin, dont une référence à l'assemblage **SelfMailer**.

En analysant le fichier **MailServerSettingsTest.cs**, vous remarquez que Visual Studio a généré une classe de tests qui comporte un membre du type `TestContext` :

```
private TestContext testContextInstance;

public TestContext TestContext
{
    get
    {
        return testContextInstance;
    }
    set
    {
        testContextInstance = value;
    }
}
```

La méthode de test `FromEmailTest` a été générée par Visual Studio. La méthode instancie un objet du type `MailServerSettings` nommé `target` puis deux variables de type `string` : `expected` et `actual`. La propriété `FromEmail` de l'objet `target` est assignée à la valeur de la variable `expected` puis la variable `actual` est assignée avec la valeur de la propriété `FromEmail` de l'objet target. Pour finir les deux variables `expected` et `actual` sont comparées dans une assertion et quel que soit le résultat, une assertion `Inconclusive` est exécutée pour signaler que le code a été généré et que le résultat n'est peut être pas concluant :

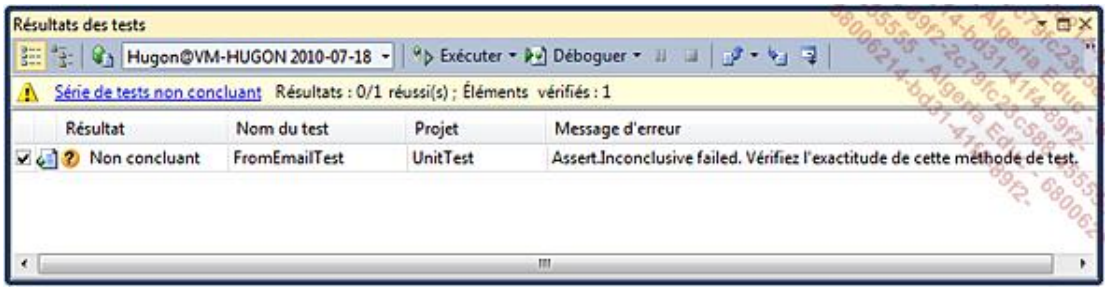
```
[TestMethod()]
public void FromEmailTest()
{
    // TODO: initialisez à une valeur appropriée
    MailServerSettings target = new MailServerSettings();
    // TODO: initialisez à une valeur appropriée
    string expected = string.Empty;
    string actual;
    target.FromEmail = expected;
    actual = target.FromEmail;
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Vérifiez l\'exactitude de cette méthode
```

```
de test.");
}
```

La classe `MailServerSettingsTest` contient également des méthodes commentées marquées avec les attributs `ClassInitialize`, `ClassCleanup`, `TestInitialize` et `TestCleanup`.

2. Le déroulement des tests

Exécutez le projet de tests depuis le menu **Test** - **Exécuter** - **Tous les tests de la solution** (**Ctrl** + **R**, **A**). Toutes les méthodes de tests sont exécutées l'une après l'autre, les résultats sont alors consultables depuis la fenêtre **Résultats des tests** :



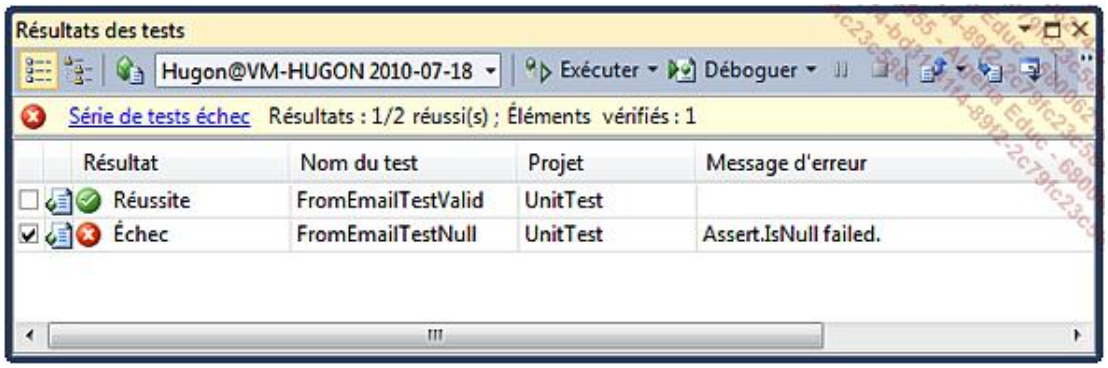
Le résultat du test est non concluant à cause de l'assertion `Inconclusive`. La colonne **Message d'erreur** donne le nom de l'assertion qui a échouée ainsi que le message qui a été passé à l'assertion.

La propriété `FromEmail` doit tester que la valeur à assigner est bien un email avant de l'affecter à la variable `fromEmail`. Si la valeur n'est pas un email valide, la valeur de la variable `fromEmail` ne doit pas être modifiée.

Modifiez la méthode de test de manière à tester ces cas :

```
[TestMethod()]
public void FromEmailTestValid()
{
    MailServerSettings target = new MailServerSettings();
    string expected = "test@abc.com";
    string actual;
    target.FromEmail = "test@abc.com";
    actual = target.FromEmail;
    Assert.AreEqual(expected, actual);
}
[TestMethod()]
public void FromEmailTestNull()
{
    MailServerSettings target = new MailServerSettings();
    target.FromEmail = "test";
    Assert.IsNull(target.FromEmail);
}
```

Exécutez les tests (**Ctrl** + **R**, **A**) et vérifiez le résultat :

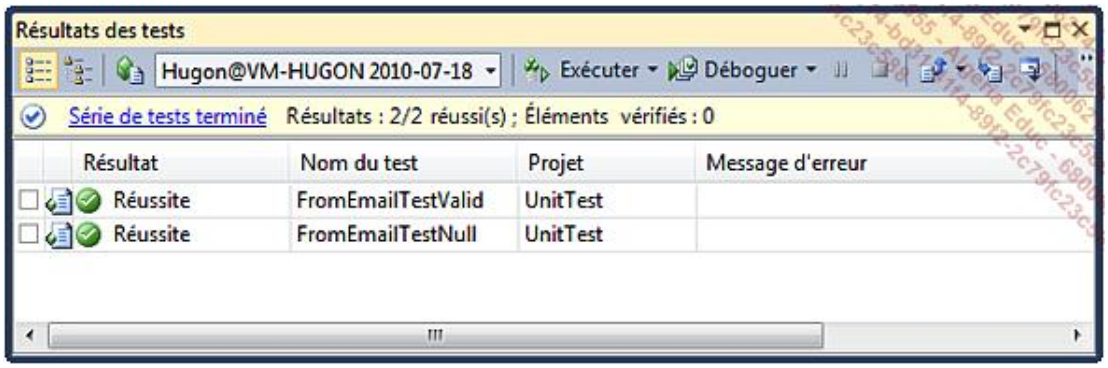


Le test de la méthode `FromEmailTestValid` est exécuté avec succès. La valeur attendue correspond bien à la valeur actuelle de la propriété `FromEmail`. Par contre, le test de la méthode `FromEmailTestNull` échoue car la valeur attendue, `string.Empty`, ne correspond pas à la valeur de la propriété `FromEmail` qui est `test`. L'accesseur set de la

propriété `FromEmail` doit donc être modifiée de manière à vérifier la validité de la valeur à assigner à la variable `fromEmail` :

```
public string FromEmail
{
   get { return this.fromEmail; }
   set
   {
      string pattern = @"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\." +
                       @"[0-9]{1,3}\.[0-9]{1,3}\.)|" +
                       @"(([a-zA-Z0-9\-]+\.)+))" +
                       @"([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
      Regex reg = new Regex(pattern);
      if (reg.IsMatch(value))
      {
         if (this.fromEmail != value)
         {
            this.fromEmail = value;
            this.HasChanged = true;
         }
      }
   }
}
```

Exécutez à nouveau les tests (**Ctrl** + **R**, **A**) et vérifiez le résultat :



Cette fois-ci les deux tests ont été un succès. On peut donc en conclure que la propriété `FromEmail` remplit son rôle.

Introduction

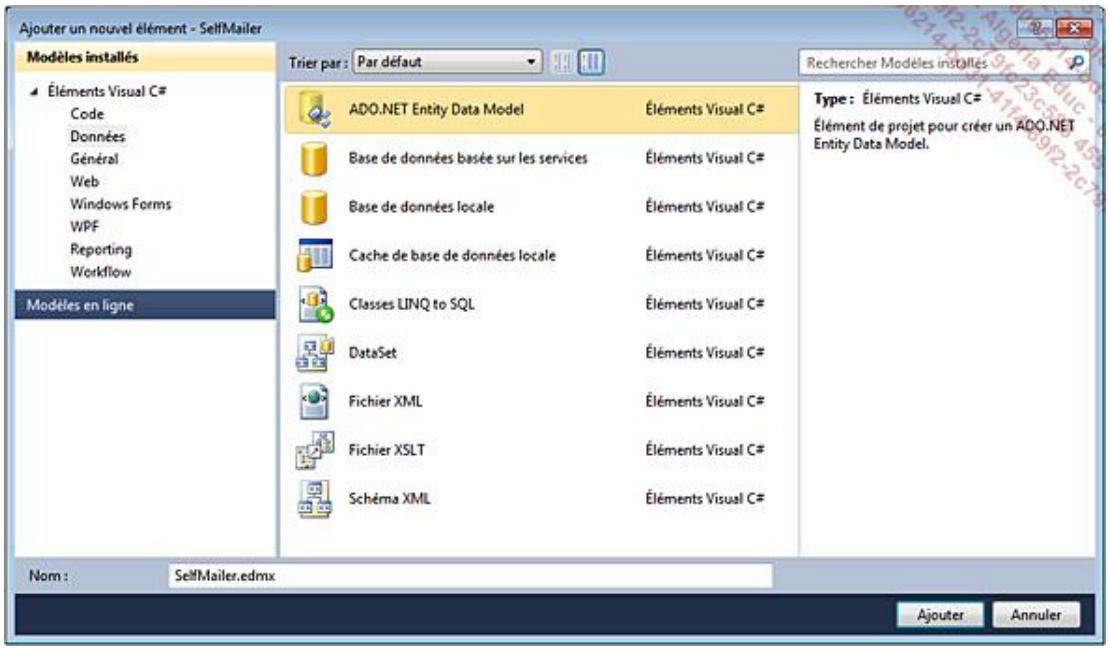
Lors de l'installation, Visual Studio installe par défaut SQL Server Express, le moteur de base de données de Microsoft dans sa version gratuite. SQL Server vous permet de créer des bases de données relationnelles pour stocker et récupérer les données de vos applications grâce au langage SQL et aux classes fournies par le Framework .NET.

Microsoft propose une solution de mappage d'objets relationnels nommée Entity Framework. Il s'agit d'une librairie permettant de créer la couche d'accès aux données (une couche d'abstraction n'obligeant plus les développeurs à accéder directement à la base de données). La communication se fait au travers d'entités définies dans un modèle EDM (*Entity Data Model*). Ce modèle est utilisé pour les opérations sur la base de données et la génération des requêtes SQL.

La création d'un modèle

Pour illustrer l'utilisation de l'Entity Framework dans le projet **SelfMailer**, nous allons créer un modèle de données permettant de stocker dans une base de données plusieurs configurations de serveurs mails et pour chacune d'elles, un ou plusieurs expéditeurs.

Ajoutez un nouveau dossier **Entities** dans le projet **SelfMailer** puis ajoutez un élément **ADO.NET Entity Data Model** nommé **SelfMailer.edmx** :



La fenêtre **Assistant EDM** s'affiche et vous propose deux choix :

- **Générer à partir de la base de données** : cette option vous permet de choisir une base de données et les objets de celle-ci qui devront être modélisés.

- **Modèle vide** : cette option crée un modèle sans entités. Elles devront être créées manuellement dans le modèle.

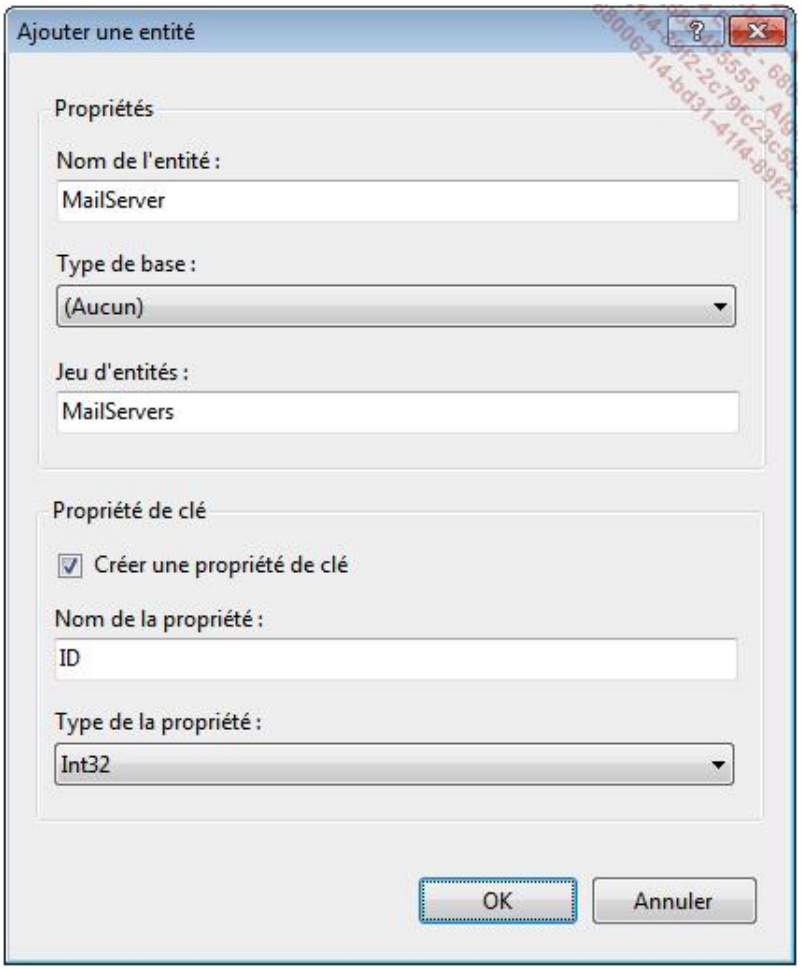
Étant donné que le projet ne contient aucune base de données, choisissez **Modèle vide** et cliquez sur le bouton **Terminer**. Visual Studio crée un fichier **SelfMailer.edmx** vide et ajoute les références aux librairies utilisées par l'Entity Framework au projet.

La création d'entités

Après sa création, Visual Studio a ouvert le fichier **SelfMailer.edmx** dans l'éditeur.

Ouvrez le menu contextuel de l'éditeur et choisissez le menu **Ajouter** - **Entité...** :

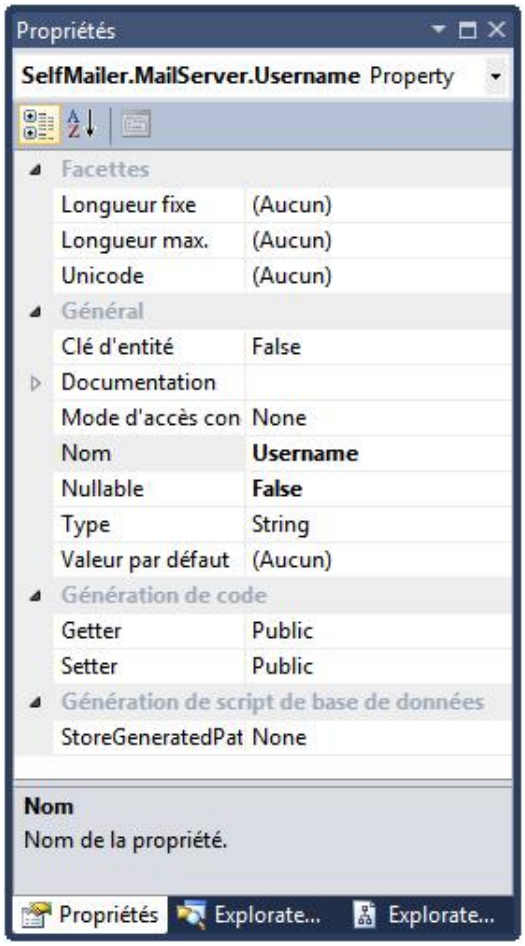
La fenêtre **Ajouter une entité** s'ouvre. Donnez le nom **MailServer** à l'entité et **MailServers** au jeu d'entités. La propriété de clé correspond à la clé primaire d'une table SQL et permet d'identifier de manière unique un enregistrement. Laissez **ID** comme nom de la propriété de clé et le type **Int32** puis cliquez sur le bouton **OK** :



Le champ **Type de base** permet de spécifier l'héritage de la nouvelle entité.

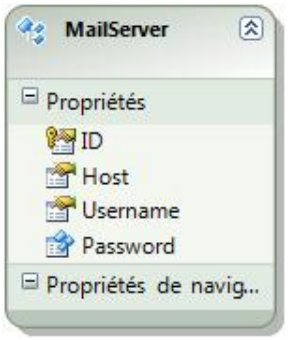
L'éditeur affiche le schéma de l'entité **MailServer** :

Pour ajouter des propriétés à une entité, ouvrez le menu contextuel de l'entité et sélectionnez le menu **Ajouter** - **Propriété scalaire**. Un nouveau champ apparaît sur le schéma. Saisissez le nom **Host**. Faites de même pour une propriété scalaire nommée **Username**. Aucune autre information n'est demandée pour la création. Les propriétés sont néanmoins modifiables à partir de la fenêtre **Propriétés**. Vous pouvez notamment définir le type de la propriété, si elle peut être null ou non, sa valeur par défaut ou encore le niveau d'accès de ses accesseurs :



Des types complexes peuvent être ajoutés à une entité. Un type complexe est une propriété contenant des sous propriétés. Ouvrez la fenêtre **Explorateur de modèles** (menu **Affichage** - **Autres fenêtres** - **Explorateur EDM**) puis ouvrez le menu contextuel du dossier **Types complexes**. Cliquez sur le menu **Créer un type complexe** et saisissez le nom **Password**. En ouvrant le menu contextuel du type complexe créé, vous pouvez ajouter des propriétés scalaires ou des propriétés complexes basées sur des types complexes. Il est ainsi possible de créer toute une hiérarchie de propriétés.

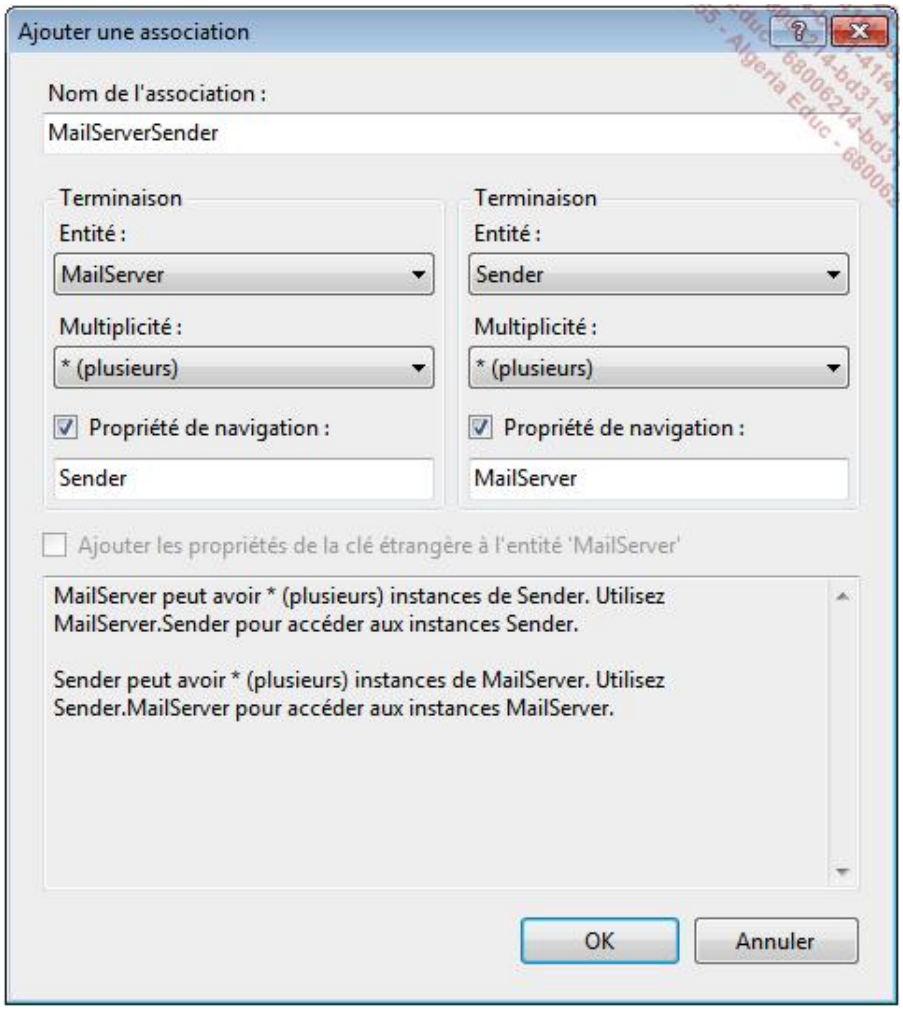
Ajoutez deux propriétés au type complexe **Password** : la première de type **String** nommée **Value** et la seconde de type **Boolean** nommée **AllowSave**. Pour finir ajoutez une propriété complexe nommée **Password** et basée sur le type complexe précédemment créé :



Ajoutez une seconde entité nommée **Sender** avec deux propriétés scalaires de type **String** nommées **Name** et **Email** :

Maintenant que nous avons des entités, nous devons les lier de manière à indiquer au modèle que ces deux entités sont associées. Une configuration de serveur mail pourra avoir de multiples expéditeurs et inversement, un expéditeur pourra être utilisé pour différentes configurations de serveurs mails. Il s'agit d'une relation de plusieurs à plusieurs.

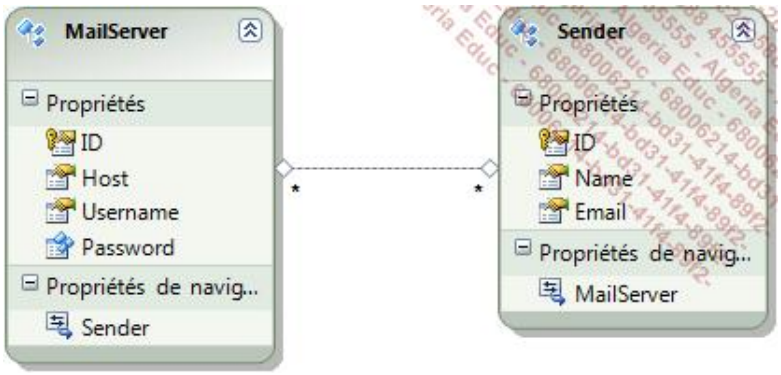
Ouvrez le menu contextuel de l'entité **MailServer** et sélectionnez le menu **Ajouter** - **Association...**. La fenêtre **Ajouter une association** s'ouvre, elle permet de définir le nom de l'association et chacune des deux entités qui seront associées ainsi que les paramètres d'associations :



La multiplicité permet de définir le rapport entre les entités : **zéro ou un**, **un** et **plusieurs**. Nous utiliserons la valeur **plusieurs** pour les deux entités. Les champs propriétés de navigation permettent de spécifier le nom de la variable qui permettra d'accéder à l'instance d'une entité à partir d'une autre, comme le spécifie l'aide en bas de la fenêtre :

```
MailServer peut avoir * (plusieurs) instances de Sender. Utilisez
MailServer.Sender pour accéder aux instances Sender.

Sender peut avoir * (plusieurs) instances de MailServer. Utilisez
Sender.MailServer pour accéder aux instances MailServer.
```

Cliquez sur le bouton **OK** pour valider l'association. Visual Studio ajoute l'association au modèle de données et indique graphiquement qu'il s'agit d'une association de plusieurs à plusieurs avec des étoiles de chaque côté de l'association :

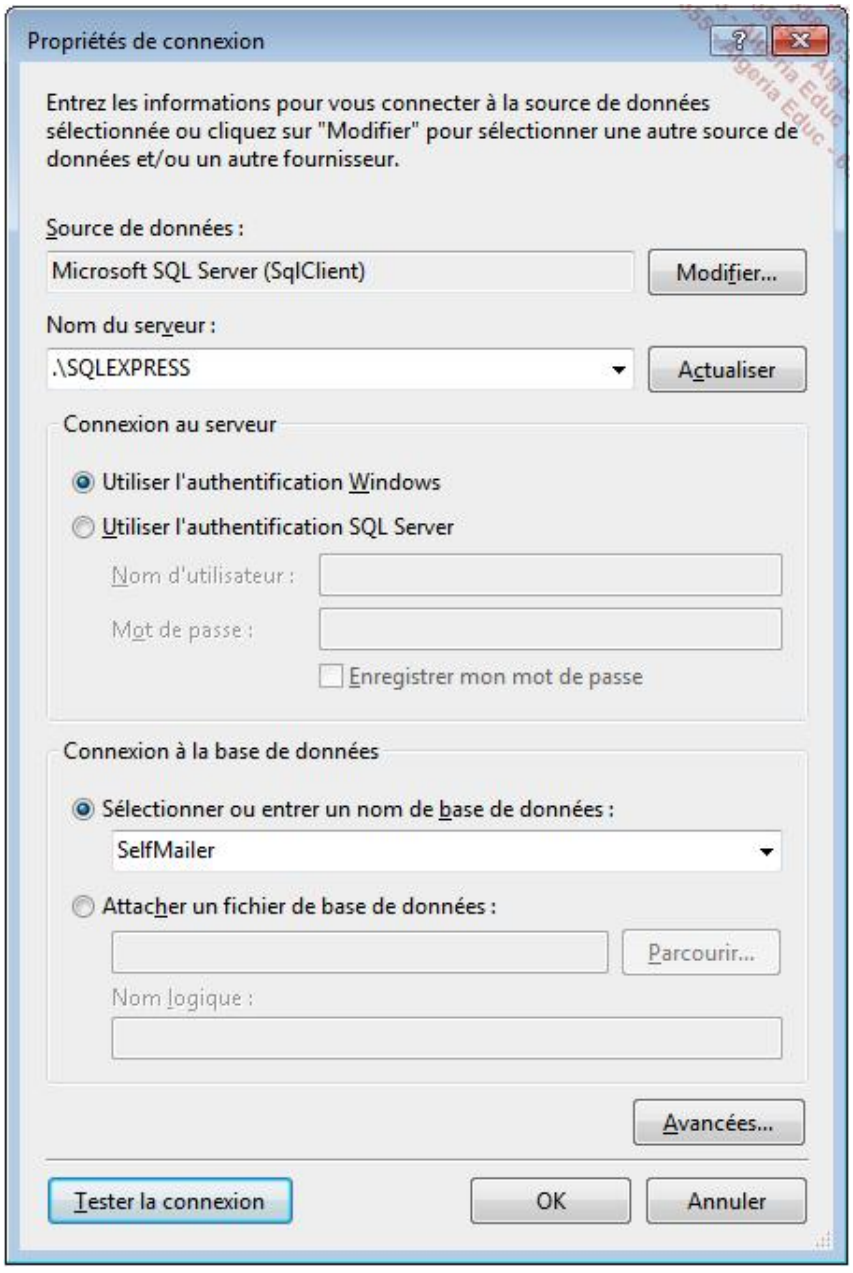


Vous pouvez remarquer également que chaque entité a une propriété de navigation supplémentaire : **Sender** pour l'entité **MailServer** et **MailServer** pour l'entité **Sender**. Ces propriétés de navigation permettent d'accéder à une instance d'entité à partir d'une autre.

La génération de la base de données

Une fois le modèle complet, il faut créer la base de données. Là encore, Visual Studio met à notre disposition un outil pour la générer.

Ouvrez le menu contextuel de l'éditeur de modèles et sélectionnez **Générer la base de données à partir du modèle...**. L'assistant de génération de la base de données s'ouvre. Cliquez sur le bouton **Nouvelle connexion...**, choisissez **Microsoft SQL Server** comme source de données et cliquez sur le bouton **Continuer**. La fenêtre **Propriétés de connexion** s'ouvre :



Lors de l'installation de Visual Studio, la version Express de SQL Server est installée par défaut sous le nom d'instance SQLEXPRESS. Saisissez donc dans le champ **Nom du serveur** la valeur **.\SQLEXPRESS** puis renseignez le nom **SelfMailer** pour le nom de la base données et cliquez sur le bouton **OK**. La base de données n'existant pas, une boîte de dialogue vous demande la confirmation de sa création.

De retour sur l'assistant de génération de la base de données, le champ de choix de la connexion est renseigné et la chaîne de connexion est affichée. Cliquez sur le bouton **Suivant**. L'assistant crée un fichier **SelfMailer.edmx.sql** contenant les instructions SQL permettant de générer la base de données :

```
SET QUOTED_IDENTIFIER OFF;
GO
USE [SelfMailer];
GO
```

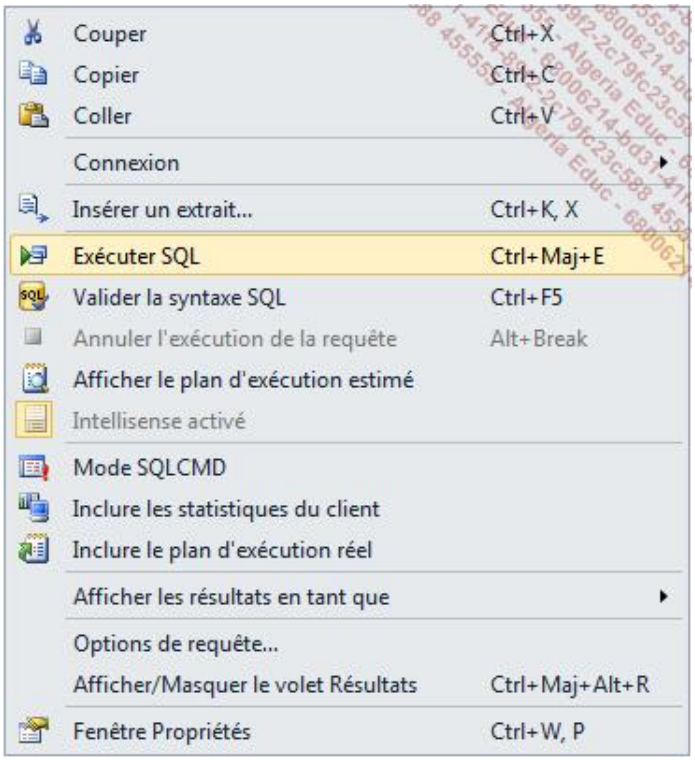
```
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- ----------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- ----------------------------------------------------


-- ----------------------------------------------------
-- Dropping existing tables
-- ----------------------------------------------------


-- ----------------------------------------------------
-- Creating all tables
-- ----------------------------------------------------

-- Creating table 'MailServers'
CREATE TABLE [dbo].[MailServers] (
    [ID] int IDENTITY(1,1) NOT NULL,
    [Host] nvarchar(max)  NOT NULL,
    [Username] nvarchar(max)  NOT NULL,
    [Password_Value] nvarchar(max)  NOT NULL,
    [Password_AllowSave] bit  NOT NULL
);
GO

-- Creating table 'Senders'
CREATE TABLE [dbo].[Senders] (
    [ID] int IDENTITY(1,1) NOT NULL,
    [Name] nvarchar(max)  NOT NULL,
    [Email] nvarchar(max)  NOT NULL
);
GO

-- Creating table 'MailServerSender'
CREATE TABLE [dbo].[MailServerSender] (
    [MailServer_ID] int  NOT NULL,
    [Sender_ID] int  NOT NULL
);
GO

-- ----------------------------------------------------
-- Creating all PRIMARY KEY constraints
-- ----------------------------------------------------

-- Creating primary key on [ID] in table 'MailServers'
ALTER TABLE [dbo].[MailServers]
ADD CONSTRAINT [PK_MailServers]
    PRIMARY KEY CLUSTERED ([ID] ASC);
GO

-- Creating primary key on [ID] in table 'Senders'
ALTER TABLE [dbo].[Senders]
ADD CONSTRAINT [PK_Senders]
    PRIMARY KEY CLUSTERED ([ID] ASC);
GO

-- Creating primary key on [MailServer_ID], [Sender_ID] in table
'MailServerSender'
ALTER TABLE [dbo].[MailServerSender]
ADD CONSTRAINT [PK_MailServerSender]
    PRIMARY KEY NONCLUSTERED ([MailServer_ID], [Sender_ID] ASC);
GO

-- ----------------------------------------------------
-- Creating all FOREIGN KEY constraints
-- ----------------------------------------------------
```

```
-- Creating foreign key on [MailServer_ID] in table
'MailServerSender'
ALTER TABLE [dbo].[MailServerSender]
ADD CONSTRAINT [FK_MailServerSender_MailServer]
    FOREIGN KEY ([MailServer_ID])
    REFERENCES [dbo].[MailServers]
        ([ID])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
GO

-- Creating foreign key on [Sender_ID] in table
'MailServerSender'
ALTER TABLE [dbo].[MailServerSender]
ADD CONSTRAINT [FK_MailServerSender_Sender]
    FOREIGN KEY ([Sender_ID])
    REFERENCES [dbo].[Senders]
        ([ID])
    ON DELETE NO ACTION ON UPDATE NO ACTION;

-- Creating non-clustered index for FOREIGN KEY
'FK_MailServerSender_Sender'
CREATE INDEX [IX_FK_MailServerSender_Sender]
ON [dbo].[MailServerSender]
    ([Sender_ID]);
GO
```

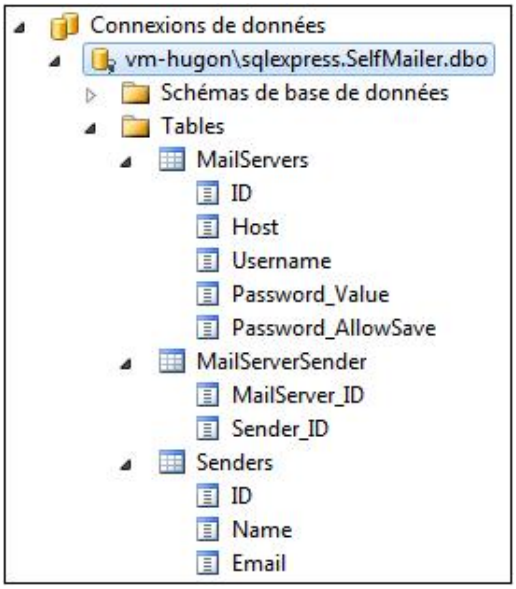
Ce script SQL commence par supprimer toutes les relations entre les tables puis les tables elles-mêmes. N'ayant pas de tables dans la base de données, ces sections sont vides. Le script continue avec la création des tables, une par entité : **MailServers** et **Senders**. Une troisième table est créée pour assurer l'association plusieurs à plusieurs entre les deux premières tables : **MailServerSender**. Les clés primaires et étrangères des tables sont ensuite ajoutées comme définies dans le modèle de données.

Remarquez que, pour la création de la propriété complexe **Password** de l'entité **MailServer**, le générateur a créé deux champs : **Password_Value** et **Password_AllowSave**.

Ouvrez le menu contextuel dans l'éditeur de script SQL et sélectionnez le menu **Exécuter SQL** (**Ctrl** + **Maj** + **E**) :



Vous pouvez ouvrir l'Explorateur de serveurs pour vérifier la création des tables :



Introduction

ADO.NET Entity Framework fournit un mappage sous la forme d'une couche d'abstraction pour obtenir un modèle d'objets basé sur une base de données référent. Les bases de données relationnelles et les langages orientés objets définissent les associations de différentes manières. Depuis la version 1.0 du framework .NET, il est possible d'utiliser le type `DataSet` qui est très similaire à la structure d'une base de données puisqu'il contient des types `DataTable`, `DataColumn`, `DataRow` et `DataRelation`. L'Entity Framework apporte un concept différent: les entités sont définies indépendamment de la structure de la base de données. Celles-ci sont alors mappées avec les tables et les relations. Le mappage entre les objets et la structure de la base de données est réalisé au travers de trois couches : la couche logique, la couche conceptuelle et la couche de mappage. Un objet de type `ObjectContext` conserve l'état des entités et leurs modifications afin de savoir quand celles-ci doivent être répercutées sur la base de données.

Le mappage

L'Entity Framework utilise plusieurs couches pour réaliser un mappage entre les tables d'une base de données et des objets. Cela peut commencer à partir d'une base de données et utiliser les outils de Visual Studio pour créer le mappage ou, comme dans le chapitre précédent, il est possible de commencer par les entités avec le concepteur de Visual Studio pour ensuite générer la base de données correspondante.

1. La couche logique

La couche logique est définie par le SSDL (*Store Schema Definition Language*) et décrit la structure des tables et des relations de la base de données.

Ouvrez le fichier **SelfMailer.edmx** créé précédemment avec un éditeur XML pour analyser son contenu. La balise `edmx:StorageModels` contient la couche logique :

```
<!-- SSDL content -->
<edmx:StorageModels>
  <Schema Namespace="SelfMailer.Store" Alias="Self"
Provider="System.Data.SqlClient" ProviderManifestToken="2008"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityS
toreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
    <EntityContainer Name="SelfMailerStoreContainer">
     ...
    </EntityContainer>
    <EntityType Name="MailServers">
     ...
    </EntityType>
    <EntityType Name="Senders">
     ...
    </EntityType>
    <EntityType Name="MailServerSender">
     ...
    </EntityType>
    <Association Name="FK_MailServerSender_MailServer">
     ...
    </Association>
    <Association Name="FK_MailServerSender_Sender">
     ...
    </Association>
  </Schema>
</edmx:StorageModels>
```

Ce code utilise le SSDL pour décrire trois tables, `MailServers`, `Senders` et `MailServerSender`. La balise `EntityContainer` contient la description de chacune des tables dans des balises `EntitySet` et les associations dans les balises `AssociationSet` :

```
<EntitySet Name="Senders"
           EntityType="SelfMailer.Store.Senders"
           store:Type="Tables" Schema="dbo" />

<AssociationSet Name="FK_MailServerSender_MailServer"
   Association="SelfMailer.Store.FK_MailServerSender_MailServer">
     <End Role="MailServer"
          EntitySet="MailServers" />
     <End Role="MailServerSender"
          EntitySet="MailServerSender" />
</AssociationSet>
```

Les balises `EntityType` définissent les colonnes des tables au travers de balises `Property` et `Key` correspondant à la clé de la table :

```
<EntityType Name="Senders">
    <Key>
      <PropertyRef Name="ID" />
    </Key>
```

```
    <Property Name="ID"
              Type="int"
              StoreGeneratedPattern="Identity"
              Nullable="false" />
    <Property Name="Name"
              Type="nvarchar(max)"
              Nullable="false" />
    <Property Name="Email"
              Type="nvarchar(max)"
              Nullable="false" />
</EntityType>
```

Les balises `Association` définissent les relations entre les tables dans les balises `End` en précisant la multiplicité (0, 1 ou plusieurs noté *) et les contraintes de clés étrangères dans la balise `ReferentialConstraint` :

```
<Association Name="FK_MailServerSender_MailServer">
    <End Role="MailServer"
         Type="SelfMailer.Store.MailServers"
         Multiplicity="1" />
    <End Role="MailServerSender"
         Type="SelfMailer.Store.MailServerSender"
         Multiplicity="*" />
    <ReferentialConstraint>
        <Principal Role="MailServer">
            <PropertyRef Name="ID" />
        </Principal>
        <Dependent Role="MailServerSender">
            <PropertyRef Name="MailServer_ID" />
        </Dependent>
    </ReferentialConstraint>
</Association>
```

2. La couche conceptuelle

La couche conceptuelle définit les types .NET qui sont décrits au sein du CSDL (*Conceptual Schema Definition Language*). La balise `edmx:ConceptualModels` du fichier **SelfMailer.edmx** créé précédemment contient la couche conceptuelle :

```
<!-- CSDL content -->
<edmx:ConceptualModels>
  <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
xmlns:cg="http://schemas.microsoft.com/ado/2006/04/codegeneration
"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityS
toreSchemaGenerator" Namespace="SelfMailer" Alias="Self"
xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/an
notation">
    <EntityContainer Name="SelfMailerEntities"
                     annotation:LazyLoadingEnabled="true">
      ...
    </EntityContainer>
    <EntityType Name="MailServer">
      ...
    </EntityType>
    <ComplexType Name="Password" >
      ...
    </ComplexType>
    <EntityType Name="Sender">
      ...
    </EntityType>
    <Association Name="MailServerSender">
      ...
    </Association>
  </Schema>
</edmx:ConceptualModels>
```

Ce code utilise le CSDL pour décrire trois types, `MailServer`, `Password` et `Sender`. La balise `EntityContainer` contient la description de chacun des types dans des balises `EntitySet` et les associations dans les balises `AssociationSet` :

```
<EntitySet Name="MailServers"
           EntityType="SelfMailer.MailServer" />

<AssociationSet Name="MailServerSender"
                Association="SelfMailer.MailServerSender">
    <End Role="MailServer" EntitySet="MailServers" />
    <End Role="Sender" EntitySet="Senders" />
</AssociationSet>
```

Les balises `EntityType` définissent les propriétés des classes au travers de balises `Property` et `Key` de la même manière que pour la couche logique avec l'ajout d'une propriété spécifique avec la balise `NavigationProperty` correspondant à l'association entre les entités `MailServer` et `Sender` :

```
<EntityType Name="MailServer">
    <Key>
        <PropertyRef Name="ID" />
    </Key>
    <Property Type="Int32"
              Name="ID"
              Nullable="false"
              annotation:StoreGeneratedPattern="Identity" />
    <Property Type="String"
              Name="Host"
              Nullable="false" />
    <Property Type="String"
              Name="Username"
              Nullable="false" />
    <Property Name="Password"
              Type="SelfMailer.Password"
              Nullable="false" />
    <NavigationProperty Name="Sender"
              Relationship="SelfMailer.MailServerSender"
              FromRole="MailServer"
              ToRole="Sender" />
</EntityType>
```

Les types complexes, comme pour le type `Password` dans l'exemple, sont définis dans des balises `ComplexType` :

```
<ComplexType Name="Password" >
    <Property Type="String" Name="Value" Nullable="false" />
    <Property Type="Boolean" Name="AllowSave" Nullable="false" />
</ComplexType>
```

Les balises `Association` définissent les relations entre les entités dans les balises `End` en précisant la multiplicité (0, 1 ou plusieurs noté *) et le type de chacune des entités associées :

```
<Association Name="MailServerSender">
    <End Type="SelfMailer.MailServer"
         Role="MailServer"
         Multiplicity="*" />
    <End Type="SelfMailer.Sender"
         Role="Sender"
         Multiplicity="*" />
</Association>
```

Visual Studio génère les types correspondant à ce schéma conceptuel dans le fichier **designer.cs** associé au modèle de données **.edmx**. Ouvrez le fichier **SelfMailer.Designer.cs** dans le dossier **Entities**. Vous pouvez constater que Visual Studio a créé les classes correspondant au schéma avec les propriétés adéquates.

3. La couche de mappage

La couche de mappage se charge de faire le lien entre la couche logique et la couche conceptuelle en utilisant le MSL (*Mapping Specification Language*).

Les informations de mappage sont contenues au sein de la balise `edmx:Mappings` :

```
<edmx:Mappings>
```

```
  <Mapping Space="C-S"
     xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
    <EntityContainerMapping
            StorageEntityContainer="SelfMailerStoreContainer"
            CdmEntityContainer="SelfMailerEntities">
      <EntitySetMapping Name="MailServers">
        ...
      </EntitySetMapping>
      <EntitySetMapping Name="Senders
        ...
      </EntitySetMapping>
      <AssociationSetMapping Name="MailServerSender"
                          TypeName="SelfMailer.MailServerSender"
                          StoreEntitySet="MailServerSender">
        ...
      </AssociationSetMapping>
    </EntityContainerMapping>
  </Mapping>
</edmx:Mappings>
```

La balise `EntityContainerMapping` contient les détails du mappage des types au sein de balises `EntitySetMapping` :

```
<EntitySetMapping Name="MailServers">
  <EntityTypeMapping TypeName="IsTypeOf(SelfMailer.MailServer)">
    <MappingFragment StoreEntitySet="MailServers">
      <ScalarProperty Name="ID" ColumnName="ID" />
      <ScalarProperty Name="Host" ColumnName="Host" />
      <ScalarProperty Name="Username" ColumnName="Username" />
      <ComplexProperty Name="Password"
                       TypeName="SelfMailer.Password">
        <ScalarProperty Name="Value"
                        ColumnName="Password_Value" />
        <ScalarProperty Name="AllowSave"
                        ColumnName="Password_AllowSave" />
      </ComplexProperty>
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

Chaque propriété scalaire d'un type est associée avec une colonne de la base de données grâce aux balises `ScalarProperty` et les types complexes sont détaillés sous forme de propriétés scalaires dans une balise `ComplexType`.

Les associations des entités et les relations des tables de la base de données sont référencées dans les balises `AssociationSetMapping` :

```
<AssociationSetMapping Name="MailServerSender"
                   TypeName="SelfMailer.MailServerSender"
                   StoreEntitySet="MailServerSender">
    <EndProperty Name="MailServer">
        <ScalarProperty Name="ID" ColumnName="MailServer_ID" />
     </EndProperty>
     <EndProperty Name="Sender">
         <ScalarProperty Name="ID" ColumnName="Sender_ID" />
     </EndProperty>
</AssociationSetMapping>
```

Travailler avec les entités

ADO.NET définit des types permettant de travailler avec les bases de données : `DbConnection`, `DbCommand` et `DbParameter` parmi d'autres. Les classes de l'Entity Framwork dérivent de ces classes de base avec notamment les classes : `EntityConnection`, `EntityCommand` et `EntityParameter`.

Le premier élément pour communiquer avec une base de données est de disposer d'une connexion. L'objet `EntityConnection` reçoit une chaîne de connexion et gère l'ouverture et la fermeture des connexions. Lors de la création du modèle de données, Visual Studio a stocké la chaîne de connexion dans le fichier de configuration de l'application **App.config** :

```
<connectionStrings>
    <add name="SelfMailerEntities"
         connectionString="metadata=
             res://*/Entities.SelfMailer.csdl|
             res://*/Entities.SelfMailer.ssdl|
             res://*/Entities.SelfMailer.msl;
             provider=System.Data.SqlClient;
             provider connection string=&amp;quot;
             Data Source=.\SQLEXPRESS;Initial Catalog=SelfMailer;
             Integrated Security=True;
             MultipleActiveResultSets=True&amp;quot;"
         providerName="System.Data.EntityClient" />
</connectionStrings>
```

Cette chaîne de connexion est composée de plusieurs éléments. L'attribut `Name` donne le nom de la connexion, il est unique. L'attribut `connectionString` donne les informations nécessaires pour se connecter à la base de données dans les parties `provider`, `provider connection string` et `Integrated Security` ainsi que sur les schémas utilisés dans la partie `metadata` :

```
metadata=res://*/Entities.SelfMailer.csdl|
         res://*/Entities.SelfMailer.ssdl|
         res://*/Entities.SelfMailer.msl;
```

Il y a une ressource référencée par schéma (logique, conceptuel et mappage).

Le dernier attribut, `providerName` permet de spécifier quel est le fournisseur qui sera utilisé.

Une chaîne de connexion peut être récupérée dans le code grâce à l'objet `ConfigurationManager` de l'espace de noms `System.Configuration` :

```
string conn = ConfigurationManager
                .ConnectionStrings["SelfMailerEntities"]
                .ConnectionString;
```

1. Les entités

Les classes des entités sont créées par Visual Studio par le concepteur d'entités comme vu dans le chapitre précédent. Ces classes dérivent du type `EntityObject`. Elles contiennent une méthode statique pour l'instanciation et les propriétés sont définies avec des accesseurs déclenchant des évènements `Changing` et `Changed` :

```
public partial class MailServer : EntityObject
{
    #region Méthode de fabrique

    public static MailServer CreateMailServer(
                global::System.Int32 id,
                global::System.String host,
                global::System.String username,
                Password password)
    {
        MailServer mailServer = new MailServer();
        mailServer.ID = id;
        mailServer.Host = host;
        mailServer.Username = username;
        mailServer.Password = StructuralObject
```

```
            .VerifyComplexObjectIsNotNull(password, "Password");
        return mailServer;
    }

    #endregion
    #region Propriétés primitives

    [EdmScalarPropertyAttribute(EntityKeyProperty=true,
                                IsNullable=false)]
    [DataMemberAttribute()]
    public global::System.Int32 ID
    {
        get { return _ID; }
        set
        {
            if (_ID != value)
            {
                OnIDChanging(value);
                ReportPropertyChanging("ID");
                _ID = StructuralObject.SetValidValue(value);
                ReportPropertyChanged("ID");
                OnIDChanged();
            }
        }
    }
    private global::System.Int32 _ID;
    partial void OnIDChanging(global::System.Int32 value);
    partial void OnIDChanged();

    ...

    #endregion
    #region Propriétés complexes

    ...

    #endregion

    #region Propriétés de navigation

    ...

    #endregion
}
```

Les propriétés complexes sont définies dans la région **Propriétés complexes**. Elles sont identiques aux propriétés primitives à part leur type qui est défini dans une classe dérivant du type `ComplexObject` comme pour la classe `Password` :

```
public partial class Password : ComplexObject
```

La région **Propriétés de navigation** contient les accesseurs assurant l'association entre les entités :

```
[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
[EdmRelationshipNavigationPropertyAttribute("SelfMailer",
                       "MailServerSender", "Sender")]
public EntityCollection<Sender> Sender
{
  get
  {
    return ((IEntityWithRelationships)this)
                .RelationshipManager
                .GetRelatedCollection<Sender>
                ("SelfMailer.MailServerSender", "Sender");
  }
  set
  {
```

```
    if ((value != null))
    {
      ((IEntityWithRelationships)this).RelationshipManager
          .InitializeRelatedCollection<Sender>
          ("SelfMailer.MailServerSender", "Sender", value);
    }
  }
}
```

Le code complet généré par Visual Studio se trouve dans le fichier **SelfMailer.Designer.cs** du dossier **Entities** dans le projet **SelfMailer**.

2. La classe ObjectContext

Pour récupérer les enregistrements de la base de données, la classe `ObjectContext` est requise. Elle gère le mappage entre les entités et les données.

La classe d'entités `SelfMailerEntities` créée par Visual Studio dérive de la classe de base `ObjectContext`. La chaîne de connexion est passée dans le constructeur en utilisant par défaut celle créée avec le modèle de données et stockée dans le fichier de configuration. Des surcharges permettent de spécifier une chaîne de connexion différente :

```
public SelfMailerEntities()
    : base("name=SelfMailerEntities", "SelfMailerEntities")
{ ... }

public SelfMailerEntities(string connectionString)
    : base(connectionString, "SelfMailerEntities")
{ ... }

public SelfMailerEntities(EntityConnection connection)
    : base(connection, "SelfMailerEntities")
{ ... }
```

Lorsque vous utilisez la surcharge prenant en paramètre un objet `EntityConnection`, l'objet se charge d'ouvrir et fermer la connexion. Si la connexion est déjà ouverte lors de son passage en paramètre au constructeur de l'objet, la connexion ne sera pas fermée automatiquement par l'objet.

La classe `SelfMailerEntities` contient également une propriété par entité retournant un objet de type `ObjectSet<TEntity>` :

```
public ObjectSet<MailServer> MailServers
{
    get
    {
        if ((_MailServers == null))
        {
            _MailServers = base
                 .CreateObjectSet<MailServer>("MailServers");
        }
        return _MailServers;
    }
}
private ObjectSet<MailServer> _MailServers;
```

La classe `ObjectContext` fournit de nombreuses fonctionnalités : elle garde la trace des objets déjà retournés depuis la base de données, elle conserve l'état des entités (ajoutées, modifiées ou supprimées), elle permet de mettre à jour les entités et de répercuter ces changements sur la base de données. La liste des méthodes suivantes donne un aperçu des fonctionnalités de la classe `ObjectContext` :

- `AcceptAllChanges` : cette méthode accepte toutes les modifications apportées aux objets dans le contexte de l'objet. Elle est implicitement appelée par la méthode `SaveChanges`.
- `AddObject` : cette méthode ajoute une nouvelle entité à la collection.
- `ApplyPropertyChanges` : si un objet est détaché du contexte puis modifié et que ces changements doivent être appliqués à l'objet du contexte, l'appel de la méthode appliquera les modifications.

- `Attach` : attache un objet au contexte, l'objet doit avoir une clé d'entité.
- `AttachTo` : contrairement à la méthode `Attach`, l'objet n'a pas la nécessité d'avoir une clé d'entité.
- `CreateDatabase` : crée la base de données en utilisant la connexion à la source de données actuelle.
- `CreateDatabaseScript` : retourne un script de langage de définition de données (DDL) qui crée des objets de schéma (tables, clés primaires, clés étrangères).
- `CreateQuery` : retourne un objet de type `ObjectQuery` pour récupérer des données.
- `DeleteObject` : cette méthode supprime un objet du contexte.
- `Detach` : cette méthode détache un objet du contexte. Ses changements ne sont alors plus surveillés.
- `GetObjectByKey` : retourne un objet qui a la clé d'entité spécifiée.
- `Refresh` : les enregistrements de la base de données peuvent être modifiés lorsqu'ils sont chargés dans le contexte. Cette méthode permet de mettre à jour le contexte. Le paramètre de type `RefreshMode` spécifie qu'elle est la version à retenir (`StoreWins` ou `ClientWins`).
- `SaveChanges` : toutes les modifications effectuées dans le contexte ne sont pas automatiquement répercutées sur la base de données. Cette méthode permet de les sauver.

3. Les relations

Les relations entre les entités sont basées sur la multiplicité. Cela peut être des relations du type de zéro ou un à un, de zéro ou un à plusieurs ou de plusieurs à plusieurs. L'Entity Framework supporte ces différents types de relations avec les concepts de table par type et de table par hiérarchie.

a. Le concept de table par type

Le concept de table par type consiste à ce qu'une table de la base de données corresponde à une entité du modèle de données. Le modèle créé précédemment utilise le concept de table par type.

Lorsque vous avez un objet de type `MailServer`, si vous souhaitez retrouver les objets de type `Sender`, il suffit d'appeler la propriété `Sender` de l'objet de type `MailServer`. Les liaisons avec la table intermédiaire `MailServerSender` sont transparentes dans le code :

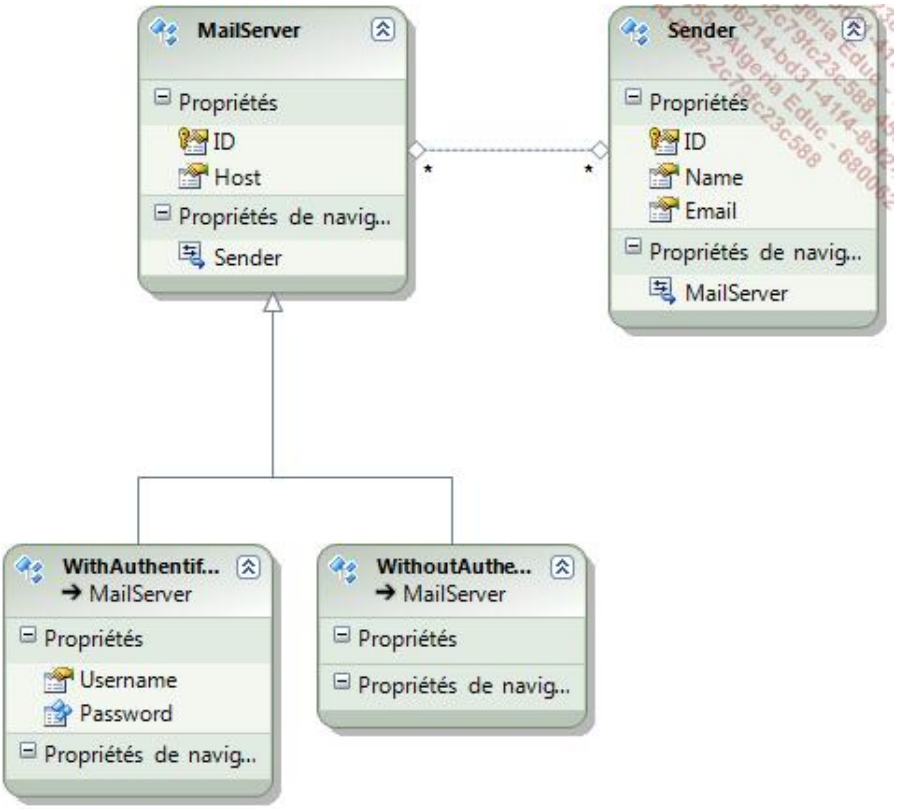
```
MailServer mailServer = new MailServer();
EntityCollection<Sender> senders = mailServer.Sender;
```

Ce mode de fonctionnement est intuitif et rapide à prendre en main.

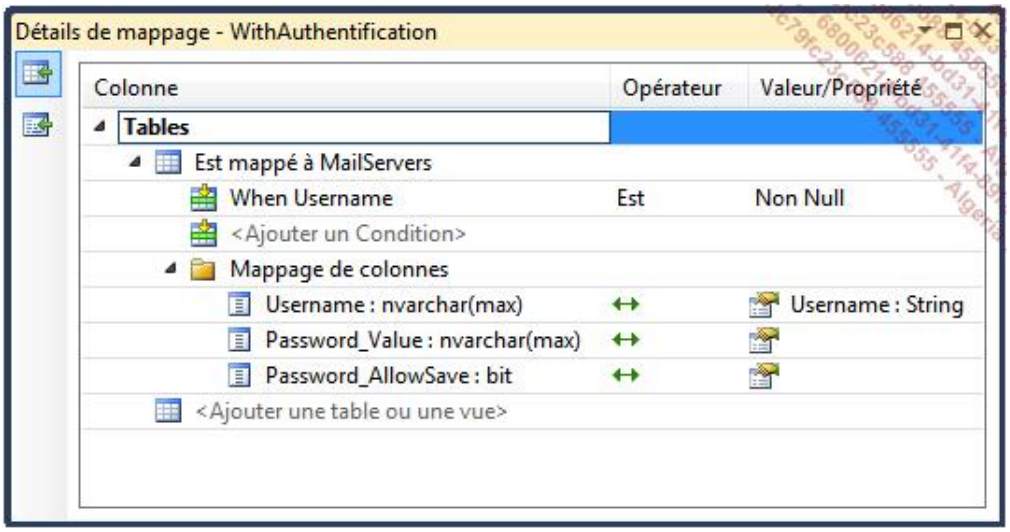
b. Le concept de table par hiérarchie

Le concept de table par hiérarchie consiste à ce qu'une table de la base de données corresponde à une hiérarchie de classes.

La table `MailServer` créée précédemment pourrait être divisée en une hiérarchie de classes. La classe de base contiendrait les propriétés `ID` et `Host`. Deux autres classes héritées contiendraient, l'une les informations de connexion (`Username`, `Password_Value` et `Password_AllowSave`) et l'autre aucune. Le schéma de la hiérarchie de classes serait le suivant :



La table `MailServer` pourra alors renvoyer des objets de type différent en fonction des valeurs stockées dans la base de données. Pour mettre en place ce mécanisme, le mappage du modèle de données doit être modifié. Visual Studio met à disposition la fenêtre **Détails de mappage** accessible depuis le menu contextuel du concepteur de modèles :

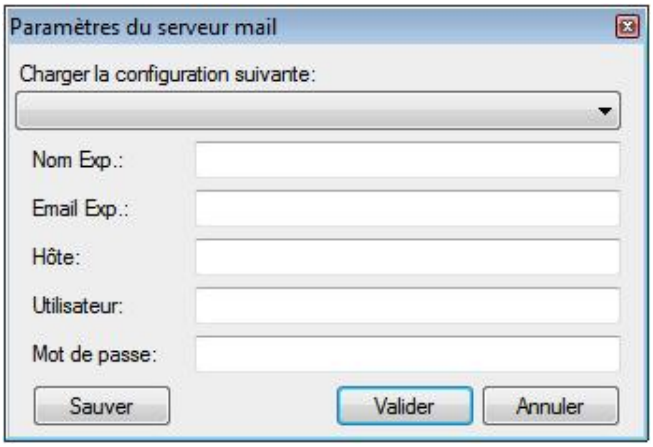


Cette fenêtre permet de spécifier une condition de mappage entre les données et les objets. Dans l'exemple précédent, un objet `WithAuthentification` est obtenu à partir d'un enregistrement de la table `MailServers` pour lequel la valeur de la colonne `Username` est non nulle.

Introduction

ADO.NET Entity Framework fournit les classes pour exécuter des requêtes et extraire les enregistrements stockés dans la base de données. Les requêtes peuvent être faites en utilisant LINQ to Entities ou Entity SQL. Le concept de LINQ sera détaillé plus loin dans l'ouvrage.

Pour illustrer ce chapitre, le formulaire **MailServerSettings** a été modifié pour afficher un contrôle ComboBox contenant les configurations stockées dans la base de données et un bouton pour sauvegarder les modifications :



Extraire les données

Les requêtes sont définies avec des objets de type `ObjectQuery` ou `ObjectQuery<T>` et doivent être exécutées au sein du contexte des entités :

```
using (SelfMailerEntities entities = new SelfMailerEntities())
{
    ...
}
```

La clause `using` est utilisée de manière à ne pas avoir à appeler la méthode `Dispose` de l'objet `SelfMailerEntities` pour libérer les ressources. Le contexte sera ainsi accessible dans la clause `using`.

1. L'extraction simple

Pour remplir le contrôle ComboBox du formulaire avec les configurations stockées dans la base de données, les entités peuvent être récupérées directement depuis le contexte :

```
ObjectQuery<MailServer> query = entities.MailServers;
```

En faisant une boucle `foreach` sur l'objet `ObjectQuery`, les entités sont disponibles :

```
foreach (MailServer mailServer in query)
{
    foreach (Sender sender in mailServer.Sender)
    {
        this.ConfigList.Items.Add(
          new ComboBoxItem(mailServer.EntityKey,
                        sender.EntityKey,
                        mailServer.Host + " - " + sender.Name));
    }
}
```

Remarquez qu'il est possible de récupérer les objets de type `Sender` liés à l'objet de type `MailServer` courant grâce à la propriété de navigation `Sender`.

La classe ComboBoxItem a été ajoutée au projet de manière à récupérer les clés de l'élément sélectionné dans le contrôle ComboBox.

2. L'extraction par clé d'entité

Le contexte permet d'accéder à un enregistrement spécifique avec la méthode `GetObjectByKey` qui prend en paramètre un objet de type `EntityKey` comme détaillé dans le gestionnaire d'évènements `ConfigList_SelectedIndexChanged` du formulaire MailServerSettings :

```
MailServer mailServers = entities
          .GetObjectByKey(result.MailServerKey) as MailServer;
```

Chaque entité possède une clé basée sur une ou plusieurs propriétés dans le modèle conceptuel. Comme pour les bases de données relationnelles, ces valeurs assurent l'unicité d'un enregistrement et améliorent la performance des requêtes. Cette clé d'entité est composée du nom de l'entité et de la valeur, correspondant à la clé primaire de la table de la base de données :

```
EntityKey eKey = new EntityKey("SelfMailerEntities.MailServer"
                               , "ID"
                               , 10);
```

Cette clé est créée automatiquement lors de l'ajout d'une entité et elle est fournie pour chaque entité extraite de la base de données.

3. L’extraction conditionnelle

La classe `ObjectQuery` expose de nombreuses méthodes permettant de créer des requêtes complexes afin d’extraire uniquement les enregistrements souhaités.

La liste suivante présente ces différentes méthodes :

- `Where` : cette méthode permet de filtrer les résultats suivant certaines conditions.

```
entities.MailServers.Where("it.Host = ’mail.domain.com’");
```

La requête créée est la suivante :

```
SELECT VALUE it
FROM (
[SelfMailerEntities].[MailServers]
) AS it
WHERE
it.Host = ’mail.domain.com’
```

- `Distinct` : cette méthode crée une requête retournant des enregistrements unique.

```
entities.MailServers.Distinct();
```

La requête créée est la suivante :

```
SET(
[SelfMailerEntities].[MailServers]
)
```

- `OrderBy` : cette méthode définit l’ordre de tri des résultats retournés par la requête.

```
entities.MailServers.OrderBy("Host, Username DESC");
```

La requête créée est la suivante :

```
SELECT VALUE it
FROM (
[SelfMailerEntities].[MailServers]
) AS it
ORDER BY
Host, Username DESC
```

- `Top` : cette méthode permet de spécifier les N premiers enregistrements qui seront retournés par la requête.

```
entities.MailServers.Top("1");
```

La requête créée est la suivante :

```
SELECT VALUE TOP(
1
) it
FROM (
[SelfMailerEntities].[MailServers]
) AS it
```

- `Select` : cette méthode retourne une projection des résultats sous la forme d’objets de type `DbDataRecord`.

```
entities.MailServers.Select("Host");
```

La requête créée est la suivante :

```
SELECT Host
FROM (
[SelfMailerEntities].[MailServers]
) AS it
```

- `Except` : cette méthode retourne un jeu de résultats sans les enregistrements retournés par la requête en paramètre de la méthode.

```
entities.MailServers.Except(
    entities.MailServers.Where("it.Host = 'mail.domain.com'"));
```

La requête créée est la suivante :

```
([SelfMailerEntities].[MailServers]
) EXCEPT (
SELECT VALUE it
FROM (
[SelfMailerEntities].[MailServers]
) AS it
WHERE
it.Host = 'mail.domain.com'
)
```

- `Union` : cette méthode combine les résultats de deux requêtes sans doublon.

```
entities.MailServers.Union(
    entities.MailServers.Where("it.Host = 'mail.domain.com'"));
```

La requête créée est la suivante :

```
([SelfMailerEntities].[MailServers]
) UNION (
SELECT VALUE it
FROM (
[SelfMailerEntities].[MailServers]
) AS it
WHERE
it.Host = 'mail.domain.com'
)
```

- `Include` : cette méthode spécifie que les enregistrements en relation doivent également être extraits de manière à ne pas avoir à faire de requêtes supplémentaires pour extraire les entités issues de la relation.

- `GroupBy` : cette méthode spécifie les critères permettant de grouper les enregistrements.

- `OfType` : cette méthode spécifie que les enregistrements extraits doivent être du type défini.

- `Skip` : cette méthode classe les résultats selon les critères spécifiés et ignore le nombre d'enregistrements spécifiés.

- `Intersect` : cette méthode retourne un jeu de résultats avec seulement les enregistrements retournés à la fois par la requête en paramètre de la méthode et par la requête courante.

- `UnionAll` : cette méthode agit de la même manière que la méthode `Union` à l'exception que les doublons sont inclus.

Le mot clé `it`, présent dans les requêtes, est automatiquement inséré en tant qu'alias de la table dont dépend la requête. Les portions de requêtes de type `string` doivent donc faire référence à ce mot clé, `it`, pour spécifier le nom de la table courante.

4. L'extraction paramétrée

La méthode `Where` de la classe `ObjectQuery` permet de filtrer les résultats. Elle prend en paramètres un objet de type `string` et des paramètres optionnels de type `ObjectParameter` :

```
entities.Senders
    .Where("it.Name = @Name && it.Email = @Email"
    , new ObjectParameter("Name", this.FromName.Text)
    , new ObjectParameter("Email", this.FromEmail.Text));
```

Le paramètre de type `string` ressemble à une requête T-SQL classique. Le mot clé `it` spécifie la table courante. Les paramètres SQL sont notés dans la requête précédés avec le caractère `@`. Les arguments suivants de type `ObjectParameter` prennent en paramètres le nom du paramètre SQL dans la requête (sans le caractère `@`) et un objet contenant la valeur à donner au paramètre SQL :

```
new ObjectParameter("Email", this.FromEmail.Text)
```

En accédant à la propriété `CommandText` de l'objet `ObjectQuery`, vous pouvez analyser la requête créée :

```
SELECT VALUE it
FROM (
[SelfMailerEntities].[Senders]
) AS it
WHERE
it.Name = @Name && it.Email = @Email
```

La requête peut également être définie directement avec la méthode générique `CreateQuery` du contexte :

```
entities.CreateQuery<MailServer>(
           "it.Name = @Name && it.Email = @Email"
           , new ObjectParameter("Name", this.FromName.Text)
           , new ObjectParameter("Email", this.FromEmail.Text));
```

Ajouter, modifier et supprimer des données

Lire, chercher, filtrer et ordonner les enregistrements de la base de données représente seulement une partie du besoin dans une application.

1. Ajouter des données

L'ajout de nouveaux enregistrements se fait en créant une nouvelle entité du type souhaité. Il suffit ensuite de renseigner ses propriétés et de l'ajouter à la collection d'entités du contexte :

```
MailServer newMailServer = new MailServer();
newMailServer.Host = this.Host.Text;
newMailServer.Username = this.Username.Text;
newMailServer.Password.Value = this.Password.Text;
newMailServer.Password.AllowSave = true;
entities.AddToMailServers(newMailServer);
```

L'ajout de la nouvelle entité à la collection du contexte peut être effectué avant ou après l'assignation des propriétés.

La nouvelle entité créée peut être liée avec une entité existante suivant les relations. Pour lier l'entité `newMailServer` précédemment créée avec une entité `Sender`, il faut l'ajouter à la collection d'entités liées :

```
Sender newSender = new Sender();
entities.AddToSenders(newSender);
newSender.MailServer.Add(newMailServer);
```

2. Modifier des données

La modification d'une entité se fait en récupérant une instance de l'entité concernée et en assignant de nouvelles valeurs à ses propriétés :

```
Sender newSender =
          entities.GetObjectByKey(selected.SenderKey) as Sender;
newSender.Name = this.FromName.Text;
newSender.Email = this.FromEmail.Text;
```

Il est également possible d'ajouter ou supprimer des relations sur une entité existante.

3. Supprimer des données

La suppression d'une entité se fait en appelant la méthode `DeleteObject` de la collection d'entités du contexte. Cette méthode prend en paramètre un objet du type de l'entité correspondant à l'entité devant être supprimée :

```
Sender delSender =
          entities.GetObjectByKey(selected.SenderKey) as Sender;
entities.Senders.DeleteObject(delSender)
```

4. L'envoi des modifications

Lorsque des opérations d'ajouts, modifications ou suppressions sont effectuées sur les collections d'entités, elles ne sont pas immédiatement répercutées sur la base de données. Le contexte garde un état de chaque entité dans la propriété `EntityState` pouvant prendre les valeurs `Added`, `Deleted`, `Modified`, `Unchanged` et `Detached`.

Cet état est géré automatiquement et permet de générer les requêtes SQL lors de l'envoi des modifications avec la méthode `SaveChanges` du contexte :

```
entities.SaveChanges();
```

Le contexte contient une propriété de type `ObjectStateManager` du même nom qui se charge de garder une trace des

entités chargées et de leurs modifications.

Cet objet contient une méthode `GetObjectStateEntries` prenant en paramètre un objet de type `EntityState` et retournant une collection d'objets de type `ObjectStateEntry` correspondant au critère spécifié :

```
IEnumerable<ObjectStateEntry> stateEntries=
    entities.ObjectStateManager
    .GetObjectStateEntries(EntityState.Added);
```

Une autre méthode de l'objet `ObjectStateManager`, `GetObjectStateEntry` prend en paramètre une entité ou une clé d'entité permettant de retrouver l'objet `ObjectStateEntry` spécifique à une entité :

```
ObjectStateEntry stateEntry =
    entities.ObjectStateManager.GetObjectStateEntry(newSender);
```

Cet objet de type `ObjectStateEntry` permet également de descendre au niveau des propriétés et d'obtenir le nom de celles qui ont été modifiées grâce à la méthode `GetModifiedProperties`. La valeur originale et la valeur courante sont alors consultables via les propriétés `OriginalValues` et `CurrentValues` de l'objet `ObjectStateEntry` :

```
foreach (string property in stateEntry.GetModifiedProperties())
{
    string original =
                stateEntry.OriginalValues[property].ToString();
    string current =
                stateEntry.CurrentValues[property].ToString();
}
```

Attacher et détacher des entités

Une entité peut être détachée de son contexte. C'est même nécessaire lorsque, par exemple, l'entité est retournée par un service Web et changée par le client. Le contexte ne peut pas savoir que l'entité a été modifiée.

La méthode `Detach` du contexte permet de détacher une entité, ses changements ne sont alors plus traqués par le contexte. Rattacher l'entité au contexte n'est pas suffisant car le contexte ne détecte pas les changements lors de l'attachement. Si l'entité originale est présente dans le contexte, elle sera utilisée mais si elle ne l'est pas ou plus, l'entité attachée sera considérée comme nouvelle. La méthode `ApplyCurrentValues` passe l'entité modifiée au contexte et si des changements ont été effectués, ils sont répercutés sur l'entité originale qui possède la même clé d'entité :

```
ComboBoxItem item = this.ConfigList.SelectedItem as ComboBoxItem;
Sender aSender =
          entities.GetObjectByKey(item.SenderKey) as Sender;

// L'entité est détachée du contexte
entities.Detach(aSender);
// Modification de l'entité
aSender.Name = "Mon nom";
// Récupération de l'original de l'entité
entities.GetObjectByKey(item.SenderKey);
// Les modifications de l'entité sont répercutées sur le contexte
entities.Senders.ApplyCurrentValues(aSender);
// Envoi des changements à la base de données
entities.SaveChanges();
```

Les requêtes LINQ

LINQ (*Language INtegrated Query*) apporte une syntaxe de requête au langage C#. Il devient possible d'accéder à différentes sources de données en utilisant une seule et même syntaxe grâce au niveau d'abstraction fourni.

1. La syntaxe

Une requête LINQ utilise des mots clé prédéfinis tels que `from`, `where`, `orderby` ou `select` pour extraire les données d'une collection d'objets :

```
var query = from ms in MailServers
            where ms.Host == "mail.mondomaine.com"
            orderby ms.Username
            select ms;
```

La requête précédente retourne une liste d'objets de type `MailServer` ayant la valeur `mail.mondomaine.com` pour la propriété `Host` et les ordonne par la propriété `Username`.

Une requête LINQ doit commencer par la clause `from` et finir par la clause `select` ou `group`. Entre ces deux clauses de début et de fin, il peut facultativement y avoir des clauses `where`, `orderby` ou `join` parmi d'autres ainsi que des clauses `from` supplémentaires.

La variable `query` contient uniquement la requête LINQ qui lui a été assignée. La requête n'est pas exécutée au moment de l'assignation mais dès que la variable est accédée dans une boucle `foreach` :

```
foreach (MailServer mailServer in query)
{
    ...
}
```

2. Les méthodes d'extension

Les méthodes d'extension ont été définies précédemment dans l'ouvrage dans le chapitre sur la création de types.

Les méthodes d'extension rendent possible l'écriture de méthodes pour une classe qui ne les définit pas. Il est également possible d'ajouter des méthodes à n'importe quelle classe qui implémente une interface spécifique de manière à ce que toutes les classes utilisent la même implémentation.

Une méthode d'extension doit être déclarée dans une classe statique et doit être définie en tant que méthode statique dont le premier paramètre est du type étendu précédé du mot clé `this`. La méthode `test` suivante étend le type `string` :

```
public static class Extension
{
    public static string test(this string s)
    {
        return s.ToLower();
    }
}
```

Il devient possible d'utiliser la méthode de la manière suivante :

```
string s1 = "ABCD";
string s2 = s1.test();  // s2 = "abcd"
```

L'utilisation de méthodes d'extension pourrait laisser penser que les règles du langage orienté objet sont mises de côté parce qu'une nouvelle méthode pour un type est définie sans changer le type ou en créer un nouveau, dérivé du type de base. Cela n'est pourtant pas le cas puisqu'une méthode d'extension ne peut pas accéder aux membres privés du type de base. Les méthodes d'extension représentent principalement une autre syntaxe pour accéder aux méthodes statiques. Elles peuvent être appelées comme une méthode classique :

```
string s1 = "ABCD";
```

```
string s2 = Extension.test(s1);
```

La classe `Enumerable` de l'espace de noms `System.Linq` définit de nombreuses méthodes d'extension. Ces méthodes d'extension étant implémentées en tant que méthodes génériques, toute collection implémentant l'interface `IEnumerable<T>` est supportée à l'image de la méthode `Where` :

```
public static IEnumerable<TSource> Where<TSource>(this
IEnumerable<TSource> source, Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            yield return item;
        }
    }
}
```

Étant donné que chacune de ces méthodes d'extension retourne un type `IEnumerable<T>`, il est possible d'invoquer les méthodes les unes après les autres en utilisant le résultat précédent pour former des requêtes complexes avec en paramètres des méthodes anonymes :

```
IEnumerable<Entities.MailServer> mailServers = MailServers
                .Where(ms => ms.Host == "mail.mondomaine.com")
                .OrderBy(ms => ms.Username)
                .Select(ms => ms);
```

Les opérateurs de requêtes

La classe `Enumerable` définit de nombreuses méthodes permettant de construire des requêtes LINQ. Chacune possède un équivalent sous forme d'un mot clé permettant de construire les requêtes LINQ sans faire appel aux méthodes.

1. Filtrer

a. Where

La méthode `Where` permet de combiner des expressions booléennes comme pour une requête SQL :

```
IEnumerable<Entities.MailServer> mailServers = MailServers
                .Where(ms => ms.Host == "mail.mondomaine.com")
                .Select(ms => ms);
```

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            where ms.Host == "mail.mondomaine.com"
            select ms;
```

Une surcharge de la méthode `Where` permet de passer en second paramètre l'index. Cet index est disponible pour chaque résultat retourné. Il peut donc être utilisé dans l'expression booléenne :

```
IEnumerable<Entities.MailServer> mailServers = MailServers
                .Where((ms, index) =>
                    ms.Host == "mail.mondomaine.com"
                    && index < 2)
                .Select(ms => ms);
```

b. OfType<TResult>

La méthode `OfType<TResult>` permet de créer un filtre sur le type de l'objet :

```
IEnumerable<Entities.MailServer> mailServers = MailServers
                .OfType<Entities.MailServer>()
                .Select(ms => ms);
```

c. SelectMany

La méthode `SelectMany` permet de sélectionner des objets à partir de plusieurs sources et le filtre peut être effectué sur ces différentes sources :

```
var mailServers = MailServers
                .SelectMany(ms => ms.Sender,
                (ms, s) => new { mailServer = ms, sender = s })
                .Where(ms => ms.sender.Name == "noreply")
                .Select(ms => ms);
```

Cette requête sélectionne tous les objets `MailServer` ayant un objet lié de type `Sender` dont la propriété `Name` a la valeur `noreply`.

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            from s in ms.Sender
            where s.Name == "noreply"
            select ms;
```

La requête LINQ est beaucoup plus lisible que l'utilisation des méthodes d'extension. Le compilateur converti la requête LINQ comportant de multiples clauses `from` en méthode d'extension `SelectMany`. Les deux exemples ci-

dessus sont donc identiques au niveau du résultat produit, seule la syntaxe est différente.

d. Skip et Take

Les méthodes `Skip` et `Take` sont utiles pour effectuer la pagination des résultats. La méthode `Skip` permet d'ignorer un certain nombre d'éléments quand à la méthode `Take`, elle permet de spécifier le nombre d'éléments à retourner :

```
int PageIndex = 2;
int PageSize = 25;
var query = (from ms in MailServers
             where ms.Host == "mail.mondomaine.com"
             select ms)
            .Skip(PageIndex * PageSize)
            .Take(PageSize);
```

Les méthodes d'extension `Skip` et `Take` sont ajoutées à la fin de la requête LINQ étant donné qu'aucun mot clé équivalent n'existe.

Les méthodes `SkipWhile` et `TakeWhile` permettent de réaliser les mêmes actions que les méthodes `Skip` et `Take` à la différence que le nombre d'éléments ignorés et gardés se fait avec un prédicat et non plus un nombre :

```
int PageIndex = 2;
int PageSize = 25;
var query = (from ms in MailServers
             where ms.Host == "mail.mondomaine.com"
             select ms)
            .SkipWhile(ms => ms.ID < PageIndex * PageSize)
            .TakeWhile(ms => ms.ID < (PageIndex + 1) * PageSize);
```

2. Ordonner

a. OrderBy

La méthode `OrderBy` a déjà été utilisée précédemment. Elle permet d'ordonner les éléments suivant le ou les critères spécifiés :

```
var mailServers = MailServers
                .OrderBy(ms => ms.Host)
                .Select(ms => ms);
```

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            orderby ms.Host
            select ms;
```

La méthode `OrderByDescending` permet d'ordonner les résultats dans l'ordre inverse de la méthode `OrderBy` :

```
var mailServers = MailServers
                .OrderByDescending(ms => ms.Host)
                .Select(ms => ms);
```

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            orderby ms.Host descending
            select ms;
```

b. ThenBy

Les méthodes `ThenBy` et `ThenByDescending` permettent de définir des ordres de tri supplémentaires dans le cas ou le premier défini par les méthodes `OrderBy` ou `OrderByDescending` comporte des éléments identiques :

```
var mailServers = MailServers
                 .OrderBy(ms => ms.Host)
                 .ThenBy(ms => ms.Username)
                 .ThenByDescending(ms => ms.Password.Value)
                 .Select(ms => ms);
```

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            orderby ms.Host,
                    ms.Username,
                    ms.Password.Value descending
            select ms;
```

Dans une requête LINQ, il suffit de spécifier les ordres de tri dans la clause `orderby`, les uns après les autres.

3. Grouper

a. GroupBy

Les résultats peuvent être groupés en se basant sur la valeur d'une clé par la méthode `GroupBy`. La méthode retourne un groupe d'objets anonymes créé :

```
var mailServers = MailServers
                 .GroupBy(ms => ms.Host)
                 .Select(g => new
                 {
                      Host = g.Key,
                      Count = g.Count()
                 });
```

La requête LINQ équivalente est la suivante :

```
var query = from ms in MailServers
            group ms by ms.Host into g
            select new
            {
                Host = g.Key,
                Count = g.Count()
            };
```

La clause `group ms by ms.Host into g` groupe tous les objets `MailServer` qui ont la même valeur pour la propriété `Host` et définit un nouvel identifiant `g` qui peut ensuite être utilisé pour accéder au groupe. Cet identifiant est utilisé dans la méthode `Select` pour créer un nouveau type anonyme contenant les propriétés `Host` et `Count`.

b. Join

La clause `join` permet de faire une jointure basée sur un critère spécifique entre deux sources :

```
var query = from ms in MailServers
            join s in Senders
                on ms.Sender.First<Entities.Sender>().ID
                equals s.ID
            select new
            {
                Host = ms.Host,
                Name = s.Name
            };
```

4. Agréger

Les méthodes d'agrégations, comme `Count`, `Sum`, `Min`, `Max`, `Average` et `Aggregate` retournent une valeur simple et non pas une séquence. Elles doivent être utilisées avec une clause `group` :

```
var query1 = from ms in MailServers
             group ms by ms.Host into g
             select new
             {
                 Host = g.Key,
                 Count = g.Count()
             };
```

Les autres méthodes s'utilisent de la même manière. `Count` retourne le nombre d'éléments, `Sum` calcule la somme d'une propriété des éléments, `Min` retourne le nombre minimum d'une propriété dans la séquence tandis que `Max` retourne le nombre maximum. `Average` calcule la moyenne d'une propriété d'une séquence. Avec la méthode `Aggregate`, il faut passer une expression Lambda qui se charge de l'agrégation des valeurs.

5. Convertir

L'exécution des requêtes LINQ est différée jusqu'à ce que l'élément soit accédé. C'est lors de l'itération que la requête est exécutée. Avec les méthodes de conversion, la requête est exécutée immédiatement et le résultat est retourné sous forme de tableau, liste ou dictionnaire :

```
List<MailServer> mailServers;
mailServers = (from ms in MailServers
               where ms.Host == "mail.mondomaine.com"
               select ms).ToList<MailServer>();
```

Les requêtes parallèles

Le Framework .NET 4 fournit la nouvelle classe `ParallelEnumerable` dans l'espace de noms `System.Linq`. Elle permet de répartir les requêtes sur plusieurs tâches de manière à améliorer leur performance.

Le gain de performances des requêtes parallèles est particulièrement visible avec des tableaux, listes ou tout type de collections de très grandes tailles. La machine qui exécute l'application doit également posséder plusieurs processeurs de manière à ce que la charge soit répartie. Les améliorations ne seront pas visibles avec une machine mono processeur.

Pour illustrer les requêtes parallèles, créez une grande liste remplie de valeurs aléatoires :

```
int size = 150000000;
List<Int64> list = new List<Int64>(size);
Random rand = new Random();
for (int i = 0; i < size; i++)
{
    list.Add(rand.Next(20));
}
```

Maintenant, ajoutez l'instruction permettant de calculer la valeur moyenne de toutes les valeurs de la liste :

```
double avg = list.AsParallel()
                 .Where(i => i > 10)
                 .Select(i => i).Average();
```

La requête LINQ équivalente est la suivante :

```
double avg = (from i in list.AsParallel()
              where i > 10
              select i).Average();
```

La seule différence avec une requête classique est l'appel à la méthode `AsParallel`. Cette méthode est définie par la classe `ParallelEnumerable` pour étendre l'interface `IEnumerable<T>`. Lors de l'exécution, la requête est exécutée au travers de plusieurs tâches. La liste est scindée en plusieurs parties de manière que chacune d'elle est traitée par une tâche pour traiter la clause `where`. Après le traitement, la liste est réunie pour effectuer le calcul de la moyenne des éléments restants.

1. Partitionner une requête

La classe `Partitioner` définie dans l'espace de noms `System.Collections.Concurrent` permet d'influencer le partitionnement des requêtes parallèles.

La méthode statique `Create` prend en paramètre un tableau ou un objet implémentant l'interface `IList<T>`. Elle permet d'influencer le parallélisme de la requête notamment avec les méthodes `WithExecutionMode`, `WithDegreeOfParallelism` et `WithMergeOptions` :

```
double avg = (from i in Partitioner.Create(list, true)
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .WithDegreeOfParallelism(10)
    .WithMergeOptions(ParallelMergeOptions.AutoBuffered)
                        where i > 10
                        select i).Average();
```

La méthode `WithExecutionMode` prend en paramètre une valeur du type `ParallelExecutionMode` qui peut être `Default` ou `ForceParallelism`.

La méthode `WithDegreeOfParallelism` prend en paramètre une valeur de type `Int32` permettant de définir le nombre de tâches maximum exécutées en parallèle.

La méthode `WithMergeOptions` prend en paramètre une valeur de type `ParallelMergeOptions` qui peut être `Default`, `NotBuffered`, `AutoBuffered` ou `FullyBuffered` permettant de déterminer le type de fusion à utiliser pour la requête. Cette valeur n'est pas forcément respectée par le système lorsque toutes les requêtes sont parallélisées.

2. Annuler une requête

La méthode `WithCancellation` peut être ajoutée à la requête parallèle en passant en paramètre un objet de type `CancellationToken` créé à partir de la classe `CancellationTokenSource`.

La requête est exécutée dans une tâche séparée qui lève une erreur de type `OperationCancelException` lorsque la requête est annulée. À partir de la tâche principale, la requête peut être annulée en invoquant la méthode `Cancel` de l'objet `CancellationTokenSource` :

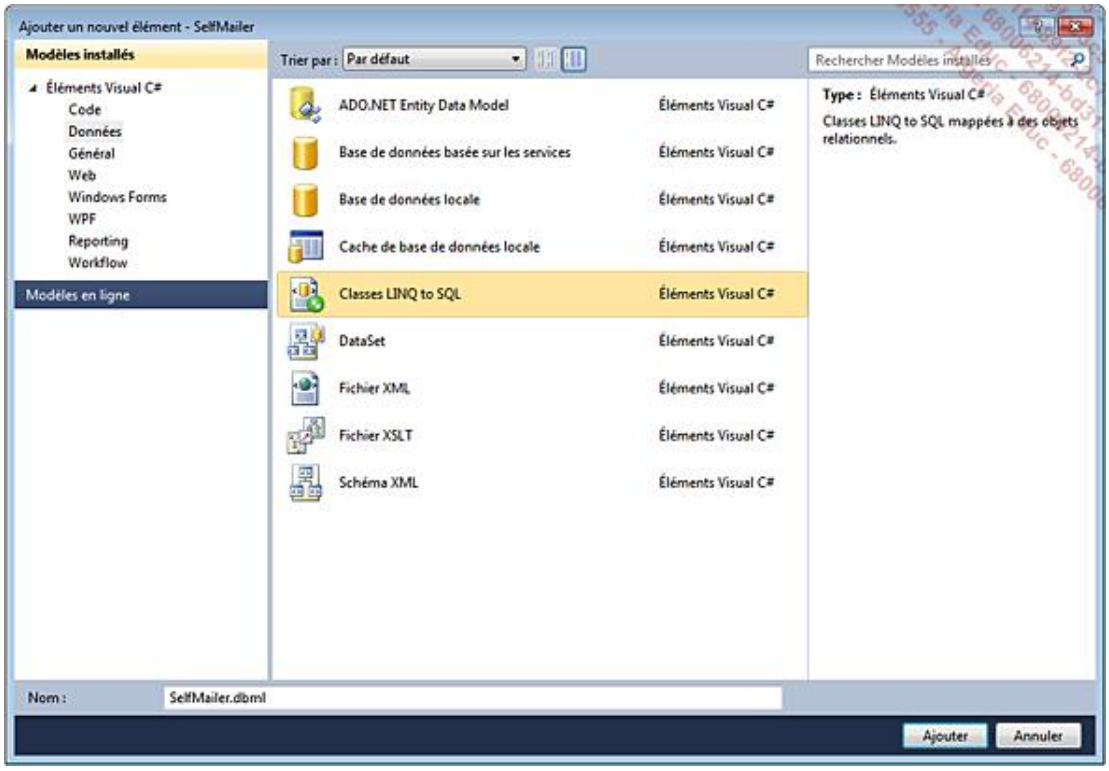
```
CancellationTokenSource ct = new CancellationTokenSource();
new Thread(() =>
{
    try
    {
        double avg = (from i in Partitioner
                              .Create(list, true).AsParallel()
                              .WithCancellation(ct.Token)
                      where i > 10
                      select i).Average();
    }
    catch (OperationCanceledException ex)
    {
        MessageBox.Show(ex.ToString());
    }
}).Start();
ct.Cancel();
```

La création de classes LINQ to SQL

LINQ to SQL est un moyen de disposer de classes fortement typées pour l'accès aux données stockées par SQL Server. LINQ utilise les relations entre les tables de la base de données pour charger les données dans des objets prêts à être utilisés par le code. LINQ to SQL n'est pas seulement utile pour l'extraction de données, il permet également de réaliser des insertions, mises à jour et suppressions de données.

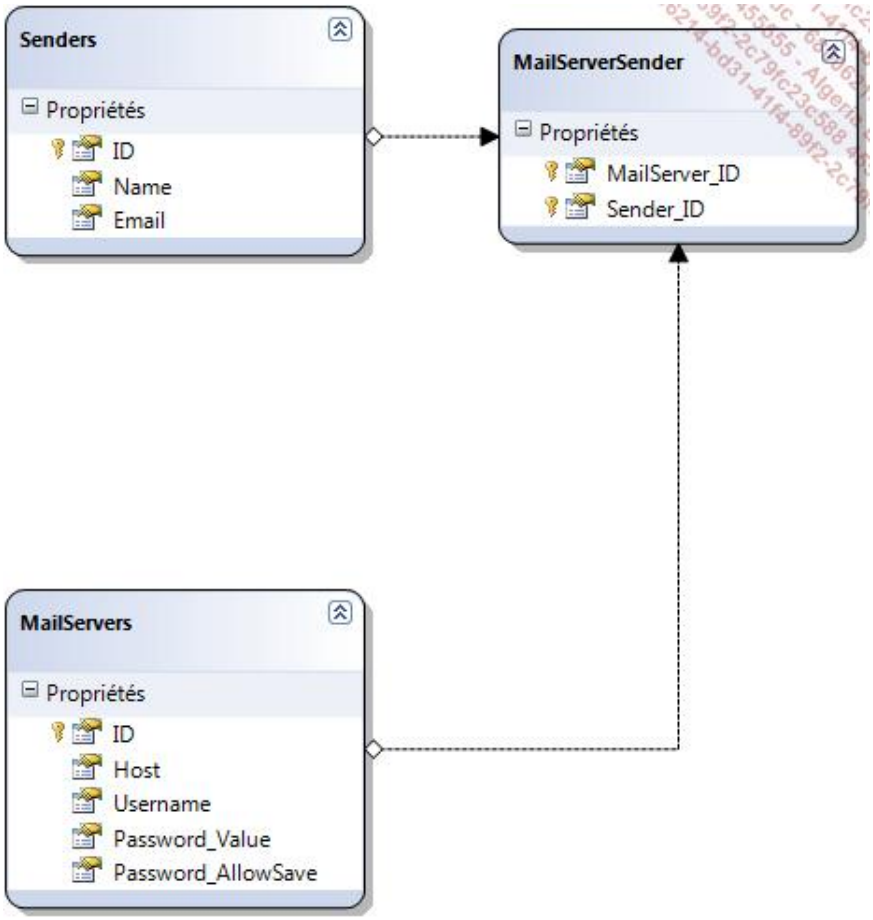
Visual Studio facilite la création de classes LINQ to SQL en fournissant un concepteur d'objets relationnels permettant de créer visuellement des objets et de définir leurs relations.

Créez un nouveau dossier nommé **LinqToSql** dans le projet **SelfMailer** et ajoutez un nouvel élément **Classes LINQ to SQL** nommé **SelfMailer.dbml** :



Visual Studio ajoute le fichier au projet et l'ouvre dans le concepteur d'objets relationnels. Les références aux librairies `System.Core`, `System.Data.DataExtensions`, `System.Data.Linq` et `System.Xml.Linq` sont également ajoutées.

Ouvrez l'**Explorateur de serveurs** et faites glisser les trois tables (MailServers, MailServerSender et Senders) de la base de données **SelfMailer** vers le concepteur d'objets relationnels :

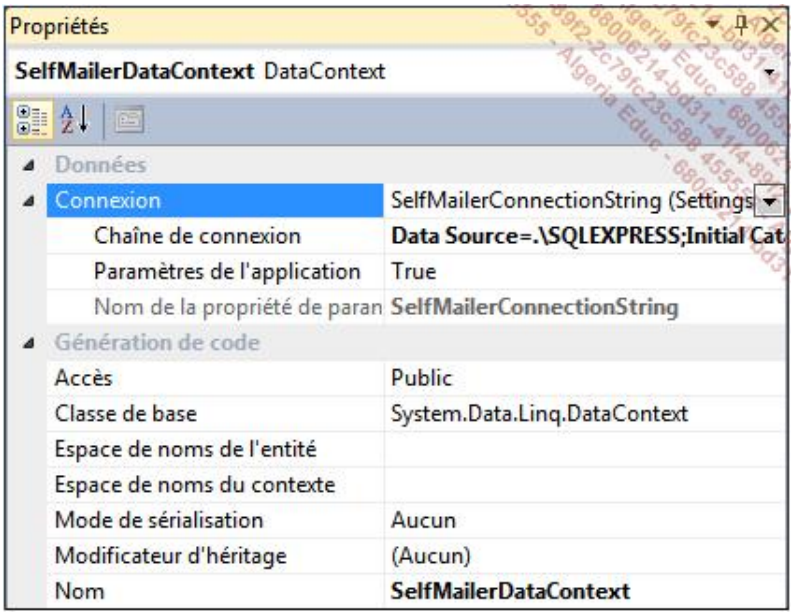


Visual Studio analyse les tables, leurs relations et leurs colonnes pour créer les classes adéquates. Ouvrez le fichier **SelfMailer.designer.cs** du dossier **LinqToSql** pour analyser le code généré automatiquement. Une classe par table a été créée ainsi qu'une dernière classe `SelfMailerDataContext` dérivant de la classe de base `DataContext`.

Tous les éléments d'une base de données possèdent leur équivalent au sein des classes LINQ to SQL. La base de données sera représentée par l'objet `DataContext`. Les tables et les vues seront représentées par des classes et des collections, les colonnes par des propriétés, les relations par des collections liées et les procédures stockées par des méthodes.

L'objet DataContext

La classe `SelfMailerDataContext` se charge des connexions avec la base de données pour l'exécution des requêtes. Elle dérive de la classe `DataContext`. Avant de pouvoir exécuter des requêtes sur une base de données, un nouvel objet `DataContext` du type désiré doit être instancié. Cette instanciation peut se faire sans paramètre. La chaîne de connexion utilisée sera celle spécifiée dans les propriétés de la classe LINQ to SQL :



L'instanciation de l'objet `DataContext` se fait de préférence au sein d'une clause `using` de manière à ce que les ressources de l'objet soient libérées :

```
// Utilisation de la clause using
using (SelfMailerDataContext context = new
SelfMailerDataContext())
{
    ...
}

// Sans clause using, l'objet doit être explicitement libéré
SelfMailerDataContext context = new SelfMailerDataContext();
...
context.Dispose();
```

1. La méthode ExecuteQuery

La manière la plus rapide pour exécuter une requête est d'utiliser la méthode `ExecuteQuery<T>` de l'objet `DataContext`. Pour récupérer tous les enregistrements d'une table, l'instruction sera la suivante :

```
IEnumerable<MailServers> query1 = context
          .ExecuteQuery<MailServers>(
                "SELECT * FROM MailServers ORDER BY Host");
```

Dans cet exemple, la méthode est appelée en passant en paramètre la requête SQL. Il est possible de spécifier des valeurs à remplacer dans la requête grâce à des chaînes de substitution au format `{i}` où `i` représente l'index du paramètre dans la méthode `ExecuteQuery<T>` :

```
foreach (MailServers mailServer in query1)
{
    IEnumerable<Senders> query2 = context
          .ExecuteQuery<Senders>(
                "SELECT S.*
                FROM Senders AS S
                INNER JOIN MailServerSender AS MSS
```

```
                ON S.ID = MSS.Sender_ID
                WHERE MSS.MailServer_ID = {0}", mailServer.ID);
}
```

Dans l'exemple précédent, la chaîne `{0}` sera remplacée par la valeur du premier paramètre de la méthode suivant la chaîne de la requête.

2. Utiliser des transactions

Vous pouvez utiliser les transactions pour effectuer une série de requêtes d'insertions, de modifications et de suppressions d'enregistrements.

Déclarez un objet de type `TransactionScope` puis exécutez les requêtes souhaitées et finissez par appeler la méthode `Complete` pour valider la transaction :

```
using (SelfMailerDataContext context = new
SelfMailerDataContext())
{
    using (TransactionScope transaction = new TransactionScope())
    {
        // Ajout, modification ou suppression d'enregistrements
        // du DataContext

        // Envoie des modifications à la base de données
        context.SubmitChanges();

        // Les enregistrements de la base de données ne sont pas
        // modifiés tant que la transaction n'est pas validée
        transaction.Complete();
     }
}
```

3. Les autres membres de DataContext

L'objet `DataContext` contient de nombreuses autres méthodes et propriétés :

Les méthodes :

- `CreateDatabase` : permet de créer une base de données sur le serveur.
- `DatabaseExists` : permet de vérifier l'existence d'une base de données et si la connexion peut être établie.
- `DeleteDatabase` : supprime une base de données.
- `ExecuteCommand` : permet d'exécuter une commande sur la base de données.
- `GetChangeSet` : permet d'accéder aux changements effectués à la base de données grâce au traçage des modifications.
- `GetTable` : retourne une collection des tables de la base de données.
- `Refresh` : permet de rafraîchir les données de l'objet par rapport à celles de la base de données.

Les propriétés :

- `ChangeConflicts` : fournit une collection d'objets qui causent des conflits concurrentiels lorsque la méthode `SubmitChanges` est appelée.
- `CommandTimeout` : permet de spécifier le délai d'attente alloué pour l'exécution d'une commande sur la base de

données.

- `Connection` : fournit la chaîne de connexion associée à la base de données.
- `DeferredLoadingEnabled` : permet de spécifier si les enregistrements en relations sont chargés ou non lors de la requête.
- `Log` : permet de spécifier où envoyer la sortie de la commande exécutée dans la requête.
- `ObjectTrackingEnabled` : spécifie si les changements des objets sont tracés ou non. Si les objets ne sont pas tracés, les modifications ne seront pas répercutées sur la base de données.

Exécuter des requêtes avec LINQ

Les requêtes LINQ permettent d'avoir une syntaxe typée qui sera évaluée lors de la compilation de l'application, contrairement aux requêtes sous forme de chaîne de caractères effectuées avec la méthode `ExecuteQuery<T>` de l'objet `DataContext` qui ne lèveront une erreur que lors de l'exécution et de la transmission de la requête à la base de données.

Les requêtes LINQ to SQL suivent les mêmes principes de la syntaxe LINQ générale.

1. Les requêtes simples

Les requêtes simples s'effectuent avec une clause `from` et une clause `select` :

```
var query = from ms in context.MailServers
            select ms;
```

Dans cet exemple, l'objet `query` est affecté avec la collection d'enregistrements de la table `MailServers`.

2. Les requêtes filtrées

La clause `where` permet de filtrer les données qui seront retournées :

```
var query = from ms in context.MailServers
            where ms.Password_AllowSave == true
            select ms;
```

Dans cet exemple, l'objet `query` est affecté avec la collection d'enregistrements de la table `MailServers` qui ont la valeur `true` pour la colonne `Password_AllowSave`.

3. Les requêtes de jointures

La jointure entre plusieurs tables est effectuée à l'aide de la clause `join` :

```
var query = from ms in context.MailServers
            join mss in context.MailServerSender
                 on ms.ID equals mss.MailServer_ID
            join s in context.Senders
                 on mss.Sender_ID equals s.ID
            select new { ms, s };
```

Dans cet exemple, l'objet `query` est affecté avec la collection d'enregistrements des tables `MailServers` et `Senders` qui sont liées. Il est possible de spécifier plusieurs clauses `join`.

Les procédures stockées

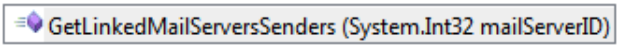
Ajoutez la procédure stockée suivante à la base de données :

```
CREATE PROCEDURE dbo.GetLinkedMailServersSenders
    (@MailServerID int = 0)
AS
    SELECT ms.*, s.*
    FROM MailServers AS ms
    INNER JOIN MailServerSender AS mss
           ON ms.ID = mss.MailServer_ID
    INNER JOIN Senders AS s
           ON mss.Sender_ID = s.ID
    WHERE ms.ID = @MailServerID
```

Cette procédure permet de récupérer les enregistrements des tables `MailServers` et `Senders` qui sont liées en fonction de l'identifiant d'un enregistrement.

1. L'ajout de procédures stockées au modèle

Pour ajouter une procédure stockée au modèle de données, il suffit de la sélectionner puis de la faire glisser depuis l'**Explorateur de serveurs** vers le concepteur d'objets relationnels :



Visual Studio ajoute une méthode du nom de la procédure stockée à l'objet dérivant de `DataContext`. Les paramètres de la méthode sont également déclarés en fonction des paramètres de la procédure stockée :

```
[global::System.Data.Linq.Mapping.FunctionAttribute
(Name="dbo.GetLinkedMailServersSenders")]
public ISingleResult<GetLinkedMailServersSendersResult>
GetLinkedMailServersSenders([global::System.Data.Linq.Mapping.Par
ameterAttribute(Name="MailServerID", DbType="Int")]
System.Nullable<int> mailServerID)
{
    IExecuteResult result = this.ExecuteMethodCall(this
           , ((MethodInfo)(MethodInfo.GetCurrentMethod()))
           , mailServerID);
    return ((ISingleResult<GetLinkedMailServersSendersResult>)
           (result.ReturnValue));
}
```

2. L'exécution de procédures stockées

L'exécution d'une procédure stockée se fait simplement en appelant la méthode adéquate du contexte et en passant en paramètres les valeurs attendues par la procédure stockée :

```
var query = context.GetLinkedMailServersSenders(1);
foreach (var item in query)
{
}
```

La boucle `foreach` après l'exécution de la méthode liée à la procédure stockée permet de parcourir les résultats de celle-ci.

Les objets XML

LINQ to XML est une implémentation du langage de requête pour les documents XML. L'espace de noms `System.Xml.Linq` expose les objets permettant de travailler avec des documents XML en mémoire de manière simple.

1. XDocument

La classe `XDocument` représente le document XML. Elle contient les membres pour accéder aux autres objets de type `XElement`, `XNamespace`, `XComment` et `XAttribute`.

Les deux méthodes les plus importantes de cette classe sont `Load` et `Save`. La méthode statique `Load` permet de charger un document XML depuis un chemin ou depuis une URL et de le stocker en mémoire, dans un objet `XDocument` :

```
XDocument xDocument = XDocument.Load(@"C:\monFichier.xml");
```

Le document XML peut également provenir d'un objet `TextReader` ou `XmlReader` et passé en argument à la méthode `Load`.

Une fois le document chargé en mémoire, il est possible de travailler avec ses propriétés :

```
string s1 = xDocument.Root.Name.ToString();
string s2 = xDocument.FirstNode.NodeType.ToString();
```

L'autre méthode importante de la classe `XDocument` est la méthode `Save` qui permet de sauvegarder le document XML en mémoire dans un fichier sur disque, un objet `TextWriter` ou un objet `XmlWriter` :

```
xDocument.Root.Add(new XAttribute("NouvelAttribut",
                                  "NouvelleValeur"));
xDocument.Save(@"C:\monFichier.xml");
```

2. XElement

L'un des types fréquemment utilisés pour travailler avec les documents XML est `XElement`. Vous pouvez créer des éléments qui sont eux-mêmes des documents XML comme créer des parties d'un document. À la création d'un élément, vous pouvez spécifier son nom, qui sera utilisé pour le nom de la balise XML, et son contenu :

```
XElement xElement = new XElement("NouvelElement",
                                 "Contenu de l'élément.");
```

Cet exemple crée le fragment XML suivant :

```
<NouvelElement>Contenu de l'élément.</NouvelElement>
```

Vous pouvez également créer des documents complets en imbriquant les objets `XElement` comme dans l'exemple suivant :

```
XElement xElement1 = new XElement("NouvelEnfant1",
                                  "Contenu de l'enfant 1.");
XElement xElement2 = new XElement("NouvelEnfant2",
                                  "Contenu de l'enfant 2.");
XElement xElement3 = new XElement("NouvelElement",
                                  xElement1,
                                  xElement2);
```

Cet exemple produit le fragment XML suivant :

```
<NouvelElement>
    <NouvelEnfant1>Contenu de l'enfant 1.</NouvelEnfant1>
    <NouvelEnfant2>Contenu de l'enfant 2.</NouvelEnfant2>
</NouvelElement>
```

3. XNamespace

L'objet `XNamespace` représente un espace de noms XML qui peut être facilement associé à un élément du document :

```
XNamespace xNamespace = "http://www.mondomaine.com/ns/xml";
XElement xElement = new XElement(xNamespace + "NouvelElement",
                                 "Contenu de l'élément.");
```

Un objet `XNamespace` est créé en lui assignant la valeur de l'espace de noms puis il est utilisé lors de la création d'un élément. Le résultat produit est le suivant :

```
<NouvelElement xmlns="http://www.mondomaine.com/ns/xml">
       Contenu de l'élément.
</NouvelElement>
```

Un espace de noms peut être attribué à tout niveau de l'arborescence du document XML :

```
XNamespace xNamespace1 = "http://www.mondomaine.com/ns/xml1";
XNamespace xNamespace2 = "http://www.mondomaine.com/ns/xml2";

XElement xElement1 = new XElement(xNamespace1 + "NouvelEnfant1",
                                  "Contenu de l'enfant 1.");
XElement xElement2 = new XElement(xNamespace2 + "NouvelEnfant2",
                                  "Contenu de l'enfant 2.");
XElement xElement3 = new XElement(xNamespace1 + "NouvelElement",
                                  xElement1,
                                  xElement2);
```

Le fragment XML produit est le suivant :

```
<NouvelElement xmlns="http://www.mondomaine.com/ns/xml1">
    <NouvelEnfant1>Contenu de l'enfant 1.</NouvelEnfant1>
    <NouvelEnfant2 xmlns="http://www.mondomaine.com/ns/xml2">
           Contenu de l'enfant 2.
    </NouvelEnfant2>
</NouvelElement>
```

Vous pouvez remarquer que l'espace de noms n'est pas appliqué à la balise `NouvelEnfant1`. La raison est que son espace de noms est identique à celui du parent. Elle en hérite donc implicitement et il n'est donc pas mentionné dans la balise.

4. XAttribute

Les éléments XML, en plus d'un contenu, peuvent avoir un ou plusieurs attributs. Ils sont représentés par la classe `XAttribute` :

```
XAttribute xAttribute = new XAttribute("MonAttribut",
                                       "ValeurAttribut");
XElement xElement = new XElement("NouvelElement",
                                 xAttribute,
                                 "Contenu de l'élément.");
```

Le fragment XML produit est le suivant :

```
<NouvelElement MonAttribut="ValeurAttribut">
    Contenu de l'élément.
</NouvelElement>
```

5. XComment

La classe `XComment` permet d'insérer facilement des commentaires au sein d'un document XML :

```
XDocument xDocument = new XDocument();
```

```
XComment xComment = new XComment("Un commentaire");
xDocument.Add(xComment);

XElement xElement = new XElement("NouvelElement",
                                 "Contenu de l'élément.");
xDocument.Add(xElement);
```

Le fragment XML produit est le suivant :

```
<!--Un commentaire-->
<NouvelElement>Contenu de l'élément.</NouvelElement>
```

Exécuter des requêtes avec LINQ

LINQ permet de réaliser des requêtes sur un document XML. Les objets présentés précédemment `XDocument`, `XElement` ou encore `XAttribute` permettent d'extraire les données souhaitées.

Les requêtes LINQ to XML suivent les mêmes principes de la syntaxe LINQ générale.

Les exemples suivants sont basés sur le fichier d'exemple LinqToXML.xml :

```
<?xml version="1.0" encoding="utf-8"?>
<SelMailer>
  <MailServer ID="1">
    <Host>mail.mondomaine.com</Host>
    <Username>monUtilisateur</Username>
    <Password>monMotDePasse</Password>
  </MailServer>
  <MailServer ID="2">
    <Host>mail.mondomaine2.com</Host>
    <Username>monUtilisateur2</Username>
    <Password>monMotDePasse2</Password>
  </MailServer>
  <Sender ID="1">
    <Name>mon nom</Name>
    <Email>email@mondomaine.com</Email>
  </Sender>
  <MailServerSender MailServerID="1" SenderID="1" />
  <MailServerSender MailServerID="2" SenderID="1" />
</SelMailer>
```

1. Les requêtes simples

Les requêtes simples s'effectuent avec une clause `from` et une clause `select` :

```
var query = from ms in xDocument.Descendants("MailServer")
            select new { Host = ms.Element("Host").Value };
```

Dans cet exemple, l'objet `query` est affecté avec la collection d'éléments enfants du document XML qui sont dans les balises `MailServer`. La requête retourne la liste des valeurs de l'élément `Host`.

2. Les requêtes filtrées

La clause `where` permet de filtrer les données qui seront retournées :

```
var query = from ms in xDocument.Descendants("MailServer")
            where ms.Attribute("ID").Value == "1"
            select new { Host = ms.Element("Host").Value };
```

Dans cet exemple, l'objet `query` est affecté avec la collection d'éléments enfants du document XML qui sont dans les balises `MailServer` et qui ont la valeur 1 pour leur attribut `ID`. La requête retourne la liste des valeurs de l'élément `Host`.

3. Les requêtes de jointures

La jointure entre plusieurs documents est effectuée à l'aide de la clause `join` :

```
var query = from ms in xDocument.Descendants("MailServer")
            join mss in xDocument.Descendants("MailServerSender")
                 on ms.Attribute("ID").Value
                 equals mss.Attribute("MailServerID").Value
```

```
            join s in xDocument.Descendants("Sender")
                 on mss.Attribute("SenderID").Value
                 equals s.Attribute("ID").Value
            select new
            {
                Host = ms.Element("Host").Value,
                Name = s.Element("Name").Value
            };
```

Dans cet exemple, l'objet `query` est affecté avec la jointure des éléments `MailServer`, `Sender` et `MailServerSender` en fonction de la valeur de leurs attributs pour réaliser la jointure. La requête retourne la liste des valeurs de l'élément `Host` des balises `MailServer` ainsi que la valeur de l'élément `Name` de la balise `Sender` associée.

Les classes de gestion du système de fichiers

Le Framework .NET fournit une série de classes dans l'espace de noms `System.IO` de la librairie **mscorlib.dll** permettant de gérer le système de fichiers au complet : les lecteurs, les dossiers et les fichiers.

1. DriveInfo

La classe `DriveInfo` fournit les membres permettant d'obtenir des informations sur les lecteurs d'une machine. Sa méthode statique `GetDrives` retourne un tableau d'objets `DriveInfo` correspondant aux lecteurs de la machine sur laquelle est exécutée l'instruction :

```
DriveInfo[] drives = DriveInfo.GetDrives();
```

La classe `DriveInfo` peut être instanciée en passant comme argument du constructeur la lettre du lecteur :

```
DriveInfo driveC = new DriveInfo("C");
```

Les informations concernant les lecteurs sont exposées par les membres de la classe `DriveInfo` :

- `AvailableFreeSpace` : indique la quantité d'espace libre sur le lecteur en octets.
- `DriveFormat` : indique le format du système de fichiers du lecteur. Cela peut être NTFS, FAT32 ou encore CDFS suivant le lecteur.
- `DriveType` : indique le type du lecteur en retournant une des valeurs de l'énumération `System.IO.DriveType` :
  - `CDRom` pour les lecteurs optiques.
  - `Fixed` pour les disques durs.
  - `Network` pour les lecteurs réseau.
  - `NoRootDirectory` pour un lecteur qui n'a pas de répertoire racine.
  - `Ram` pour un lecteur RAM.
  - `Removable` pour les lecteurs de disquettes ou les disques durs externes.
  - `Unknow` lorsque le type du lecteur est inconnu.
- `IsReady` : cette propriété booléenne indique si le lecteur est prêt à être utilisé.
- `Name` : indique le nom du lecteur, il s'agit de sa lettre d'accès.
- `RootDirectory` : indique le chemin de la racine du lecteur en retournant un objet `DirectoryInfo`.
- `TotalFreeSpace` : indique la quantité totale d'espace libre sur le lecteur en octets.
- `TotalSize` : indique la quantité totale d'espace du lecteur.
- `VolumeLabel` : retourne le nom de volume d'un lecteur. Par exemple Disque local.

À l'exception de la propriété `VolumeLabel`, elles sont toutes en lecture seule.

2. Directory et DirectoryInfo

La classe `Directory` est statique. Elle est utilisée en fournissant le chemin du dossier lors de l'appel d'une méthode statique. Si vous souhaitez n'effectuer qu'une seule action sur un dossier, utilisez cette classe de manière à économiser l'instanciation d'un objet. La classe `DirectoryInfo` implémente pratiquement les mêmes méthodes que la classe `Directory`. Si plusieurs opérations doivent être exécutées sur un même dossier, la classe `DirectoryInfo` sera plus efficace car les informations sur le dossier seront lues une seule fois lors de l'instanciation, quel que soit le nombre d'opérations effectuées sur le dossier.

La plupart des méthodes de la classe `DirectoryInfo` sont implémentées dans la classe `Directory`, il est possible de créer, supprimer et modifier les propriétés d'un dossier grâce à ces classes :

```
// Création d'un dossier avec la classe statique Directory
Directory.CreateDirectory(@"C:\dossier");

// Création d'un dossier avec la classe DirectoryInfo
DirectoryInfo directory = new DirectoryInfo(@"C:\dossier");
directory.Create();
```

L'utilisation de la classe statique `Directory` pour une opération sur un dossier n'oblige pas à l'instanciation d'un objet. Si le dossier à créer est déjà existant, aucune exception ne sera levée. Le second exemple qui utilise la classe `DirectoryInfo` oblige l'instanciation d'un objet, ce qui est légèrement plus long. L'avantage est que l'objet est prêt pour effectuer de multiples opérations.

Lors de l'instanciation d'un objet `DirectoryInfo`, si le chemin passé en paramètre n'existe pas, aucune exception n'est levée. Elle le sera lors du premier appel de méthode (à l'exception de la méthode `Create`). Pour vérifier l'existence d'un dossier, la classe `DirectoryInfo` expose la propriété booléenne `Exists` :

```
DirectoryInfo directory = new DirectoryInfo(@"C:\dossier");
if (directory.Exists)
{
    directory.Delete();
}
```

En cas de tentative d'opération sur un dossier non existant, une exception du type `DirectoryNotFoundException` sera levée.

La classe statique `Directory` réagit de manière semblable lorsqu'une opération est tentée sur un dossier inexistant. Elle expose une méthode `Exists` prenant en paramètre le chemin du dossier et retourne une valeur booléenne déterminant l'existence ou non du dossier :

```
if (Directory.Exists(@"C:\dossier"))
{
    Directory.Delete(@"C:\dossier");
}
```

Les classes `Directory` et `DirectoryInfo` exposent plusieurs autres méthodes permettant d'effectuer les opérations communes aux dossiers dont :

- Le déplacement de dossier se fait avec la méthode `Move` de la classe `Directory` ou `MoveTo` de la classe `DirectoryInfo` :

```
if (Directory.Exists(@"C:\dossier"))
{
    Directory.Move(@"C:\dossier", @"C:\NouveauDossier");
}

DirectoryInfo directory = new DirectoryInfo(@"C:\dossier");
if (directory.Exists)
{
    directory.MoveTo(@"C:\NouveauDossier");
}
```

- L'obtention des sous-dossiers est réalisée par l'appel des méthodes `GetDirectories` ou `EnumerateDirectories` présentent dans les deux classes `Directory` et `DirectoryInfo` :

```
string[] dirs = Directory.GetDirectories(@"C:\dossier");
IEnumerable<string> enumDirs = Directory
```

```
                        .EnumerateDirectories(@"C:\dossier");
DirectoryInfo directory = new DirectoryInfo(@"C:\dossier");
DirectoryInfo[] dirs = directory.GetDirectories();
IEnumerable<DirectoryInfo> enumDirs = directory
                                    .EnumerateDirectories();
```

Alors que les méthodes `GetDirectories` et `EnumerateDirectories` de la classe `Directory` retournent respectivement un tableau de type `string` ou un objet `IEnumerable` générique de type `string`, la classe `DirectoryInfo` retourne des objets du type `DirectoryInfo`.

Ces méthodes comportent des surcharges permettant de spécifier un modèle de nom de dossier afin de filtrer les résultats. Une autre surcharge permet également de spécifier si la recherche doit être effectuée seulement dans le dossier courant ou également dans ses sous-dossiers grâce à une énumération du type `System.IO.SearchOption` :

```
string[] dirs = Directory.GetDirectories(@"C:\",
                            "m?o",
                            SearchOption.TopDirectoryOnly);
string[] dirs = Directory.GetDirectories(@"C:\",
                            "m*",
                            SearchOption.AllDirectories);
```

Le modèle peut contenir les caractères `*` et `?`. Le caractère `*` remplace un ou plusieurs autres caractères tandis que le caractère `?` remplace un seul et unique caractère.

- L'obtention des fichiers contenus dans un dossier s'effectue sur le même modèle que pour les sous-dossiers avec les méthodes `GetFiles` ou `EnumerateFiles` présentent dans les deux classes `Directory` et `DirectoryInfo`.

3. File et FileInfo

Les classes `File` et `FileInfo` ont un mode de fonctionnement similaire aux classes `Directory` et `DirectoryInfo` dans le cadre des fichiers. La classe `File` est statique tandis que la classe `FileInfo` doit être instanciée. Le choix de l'utilisation d'une classe ou de l'autre dépend du nombre d'opérations à effectuer sur un fichier. Une opération unique sera plus rapidement réalisée avec la classe `File` alors qu'une multitude d'opérations entraînera l'utilisation d'un objet `FileInfo` pour l'amélioration des performances.

Ces classes permettent, d'une part, de créer, déplacer, supprimer des fichiers ainsi que de modifier leurs propriétés et, d'autre part, de lire et écrire dans les fichiers :

```
FileStream fileStream = File.Create(@"C:\fichier.txt");
// Ecriture dans le fichier
fileStream.Dispose();
```

```
FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
FileStream fileStream = fileInfo.Create();
// Ecriture dans le fichier
fileStream.Dispose();
```

La création d'un fichier retourne un objet du type `FileStream` associé. L'écriture de données dans ce flux entraînera l'écriture dans le fichier. L'appel de la méthode `Create` sur un fichier déjà existant entraîne l'écrasement de toutes les données existantes. Pour lire ou écrire des données dans un fichier existant, il faut utiliser les méthodes `Open`, `OpenRead`, `OpenText` ou `OpenWrite` :

```
FileStream fs = File.Open(@"C:\fichier.txt", FileMode.Open);
// Opérations sur le fichier
fs.Dispose();

FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
FileStream fileStream = fileInfo.Open(FileMode.Open);
// Opérations sur le fichier
fileStream.Dispose();
```

Le paramètre de type `FileMode` permet de spécifier le mode d'ouverture du fichier. L'énumération contient les valeurs suivantes :

- `CreateNew` : un nouveau fichier sera créé. Si le fichier est déjà existant, une exception du type `IOException`

sera levée.

- `Create` : un nouveau fichier sera créé. Si le fichier existe déjà, il sera remplacé. Cela équivaut à écrire que si le fichier n'existe pas, il est ouvert avec le mode `CreateNew` et s'il existe il sera ouvert avec le mode `Truncate`.
- `Open` : spécifie que le fichier doit uniquement être ouvert. S'il n'existe pas, une exception du type `FileNotFoundException` est levée.
- `OpenOrCreate` : indique que si le fichier est existant, alors il doit être ouvert et dans le cas contraire, il doit être créé.
- `Truncate` : indique que le fichier doit être ouvert et vidé de son contenu. Lorsqu'un fichier est ouvert dans ce mode, il ne peut pas être lu.
- `Append` : ce mode ouvre le fichier ou le crée s'il est inexistant et le flux est positionné à la fin de celui-ci. Le fichier ne pourra pas être lu sous peine de lever une exception du type `NotSupportedException`.

La lecture et l'écriture de données dans les fichiers seront étudiées dans la section suivante.

Comme pour les dossiers, les classes `File` et `FileInfo` exposent un membre `Exists` sous forme de propriété pour la classe `FileInfo` et sous forme de méthode pour la classe `File` :

```
if (File.Exists(@"C:\fichier.txt"))
{
    // ...
}

FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
if (fileInfo.Exists)
{
    // ...
}
```

Les classes `File` et `FileInfo` exposent plusieurs autres méthodes permettant d'effectuer les opérations communes aux dossiers dont :

- La suppression de fichiers est effectuée à l'aide de la méthode `Delete` présente dans les deux classes, `File` et `FileInfo` :

```
File.Delete(@"C:\fichier.txt");

FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
fileInfo.Delete();
```

  Si le fichier qui est tenté d'être supprimé n'existe pas, aucune exception n'est levée.

- Le déplacement ou la copie de fichiers se fait avec les méthodes `Copy` et `Move` de la classe `File` et les méthodes `CopyTo` et `MoveTo` de la classe `FileInfo` :

```
File.Copy(@"C:\fichier.txt", @"C:\NouveauFichier.txt");
File.Move(@"C:\fichier.txt", @"C:\NouveauFichier.txt");

FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
fileInfo.CopyTo(@"C:\NouveauFichier.txt");
fileInfo.MoveTo(@"C:\NouveauFichier.txt");
```

  Toute tentative de copier ou déplacer un fichier vers un emplacement qui contient un fichier du même nom entraîne la levée d'une exception de type `IOException`.

- L'encryptage et le décryptage de fichiers sont réalisés avec les méthodes `Encrypt` et `Decrypt`, le cryptage est effectué de telle sorte que seul le compte utilisé pour chiffrer le fichier puisse le déchiffrer :

```
File.Encrypt(@"C:\fichier.txt");
File.Decrypt(@"C:\fichier.txt");

FileInfo fileInfo = new FileInfo(@"C:\fichier.txt");
fileInfo.Encrypt();
fileInfo.Decrypt();
```

Un fichier crypté avec ces méthodes est remarquable dans l'explorateur de fichiers Windows car son nom est écrit en vert.

4. Path

La classe `Path` est statique, elle expose des méthodes permettant de réaliser des opérations sur les chemins des dossiers et des fichiers. Ces opérations ne sont pas physiquement répercutées sur les fichiers et dossiers :

- `ChangeExtensions` : cette méthode permet de modifier l'extension d'un fichier. Le premier paramètre doit être le chemin du fichier et le second la nouvelle extension :

```
string result = Path.ChangeExtension(@"C:\fichier.txt", "doc");
// result = "C:\fichier.doc"
```

- `Combine` : cette méthode permet de combiner plusieurs chemins :

```
string result = Path.Combine(@"C:\Dossier1",
                             "Dossier2",
                             "fichier.txt");
// result = "C:\Dossier1\Dossier2\fichier.txt"
```

Il est plus efficace d'utiliser la méthode `Combine` plutôt que de réunir les différentes parties d'un chemin en spécifiant le caractère séparateur qui est différent d'un OS à l'autre.

- `GetDirectoryName` : retourne le chemin du répertoire relatif au chemin spécifié :

```
string result =
Path.GetDirectoryName(@"C:\Dossier1\fichier.txt");
// result = "C:\Dossier1"
```

- `GetExtension` : retourne l'extension du fichier dans le chemin spécifié :

```
string result = Path.GetExtension(@"C:\Dossier1\fichier.txt");
// result = ".txt"
```

- `GetFileName` : retourne le nom du fichier avec son extension :

```
string result = Path.GetFileName(@"C:\fichier.txt");
// result = "fichier.txt"
```

- `GetFileNameWithoutExtension` : retourne le nom du fichier sans son extension :

```
string result =
Path.GetFileNameWithoutExtension(@"C:\fichier.txt");
// result = "fichier"
```

- `GetFullPath` : retourne le chemin absolu du chemin relatif spécifié :

```
string result = Path.GetFullPath(@"fichier.txt");
```

Le chemin par défaut est celui d'exécution de l'application.

- `GetInvalidFileNameChars` : retourne un tableau des caractères non autorisés dans les noms des fichiers.
- `GetInvalidPathChars` : retourne un tableau des caractères non autorisés dans les chemins.
- `GetPathRoot` : retourne le répertoire racine du chemin spécifié :

```
string result = Path.GetPathRoot(@"C:\Dossier1\fichier.txt");
// result = "C:\"
```

- `GetRandomFileName` : retourne un nom de fichier aléatoire :

```
string result = Path.GetRandomFileName();
// result = "0zyso02u.zgv"
```

- `GetTempFileName` : crée un fichier temporaire vide nommé de manière unique et retourne le chemin complet de celui-ci :

```
string result = Path.GetTempFileName();
// result = "C:\Users\Hugon\AppData\Local\Temp\tmp7032.tmp"
```

- `GetTempPath` : retourne le chemin du dossier temporaire du système :

```
string result = Path.GetTempPath();
// result = "C:\Users\Hugon\AppData\Local\Temp\"
```

- `HasExtension` : retourne une valeur booléenne indiquant si le chemin spécifié contient une extension de fichier :

```
bool result = Path.HasExtension(@"C:\Dossier1\fichier.txt");
// result = true
bool result = Path.HasExtension(@"C:\Dossier1");
// result = false
```

- `IsPathRooted` : retourne une valeur booléenne indiquant si le chemin d'accès est relatif ou absolu :

```
bool result = Path.IsPathRooted(@"C:\Dossier1");
// result = true
bool result = Path.IsPathRooted(@"Dossier1");
// result = false
```

Travailler avec le système de fichiers

1. Les objets Stream

Les objets `Stream` sont utilisés pour le transfert de données soit entre une source externe et l'application (il s'agit alors d'une lecture de données), soit entre l'application et une source externe (il s'agit d'écriture de données).

La source externe d'un flux peut provenir d'un fichier, d'un emplacement mémoire ou encore du réseau. Suivant la source, un objet différent sera utilisé. Par exemple pour un flux en mémoire, on utilisera un objet de type `System.IO.MemoryStream` tandis que pour le transfert de données au travers d'un protocole réseau, on utilisera la classe `System.IO.NetworkStream`. Lorsque vous travaillez avec des fichiers, les classes utilisées seront, d'une part, `FileStream` pour écrire et lire des données binaires en particulier et pour tout type de fichier en général. D'autre part, les classes `StreamReader` et `StreamWriter` ont spécialement été conçues pour lire et écrire dans des fichiers texte.

L'avantage d'utiliser un objet distinct pour le transfert de données est de pouvoir rendre plus facile le changement du type de la source externe. En conservant la séparation entre le code de l'application et le concept de la source de données particulière, le code est plus facilement réutilisable.

2. La classe FileStream

La classe `FileStream` est utilisée pour lire et écrire des fichiers binaires. Son constructeur et ses surcharges peuvent comprendre jusqu'à quatre paramètres permettant de déterminer le fichier, le mode d'ouverture, le type d'accès et le type de verrouillage :

```
FileStream stream = new FileStream(@"C:\fichier.dat",
                                   FileMode.OpenOrCreate,
                                   FileAccess.ReadWrite,
                                   FileShare.None);
```

L'énumération `FileMode` permet de spécifier la façon dont doit être ouvert le fichier. Les valeurs possibles ont été présentées plus tôt dans ce chapitre.

L'énumération `FileAccess` détermine quels types d'opérations pourront être effectués sur le fichier. Les valeurs sont `Read` pour ouvrir le fichier en lecture seule, `Write` pour l'ouvrir en écriture seule et `ReadWrite` pour l'ouvrir en mode lecture et écriture.

L'énumération `FileShare` permet de définir le type d'accès des autres objets sur le fichier. Vous pouvez spécifier une ou plusieurs valeurs parmi :

- `None` pour refuser tout partage.
- `Read` pour autoriser l'ouverture en lecture du fichier.
- `Write` pour autoriser l'ouverture du fichier en écriture.
- `ReadWrite` pour autoriser l'ouverture du fichier en lecture et écriture.
- `Delete` pour permettre de supprimer le fichier.
- `Inheritable` pour créer un handle de fichier hérité par les processus enfants.

Les valeurs de cette énumération peuvent être combinées avec l'opérateur `|` :

```
FileStream stream = new FileStream(@"C:\fichier.dat",
                                   FileMode.OpenOrCreate,
                                   FileAccess.ReadWrite,
                                   FileShare.Read |
                                   FileShare.Delete);
```

Les objets de type `FileStream` peuvent aussi être instanciés à partir de la classe statique `File` grâce aux méthodes `Open`, `OpenRead` et `OpenWrite` qui ouvrent le fichier spécifié respectivement en mode lecture et écriture, lecture seule et écriture seule :

```
FileStream stream = File.Open(@"C:\fichier.dat",
FileMode.Create);
FileStream stream = File.OpenRead(@"C:\fichier.dat");
FileStream stream = File.OpenWrite(@"C:\fichier.dat");
```

La classe `FileInfo` contient les méthodes identiques.

Il existe deux méthodes pour lire des données. La première est la méthode `ReadByte` qui prend le premier bit à partir de la position courante du flux et le transforme en type `int`. Si la fin du flux est atteinte, la valeur -1 est retournée :

```
int nextByte = stream.ReadByte();
```

La seconde méthode de lecture des données est `Read`. Elle permet de lire un nombre déterminé de bits et de les placer dans un tableau de type `byte`. Elle retourne le nombre de bits effectivement lus :

```
byte[] bytes = new byte[10];
int bytesRead = stream.Read(bytes, 0, 10);
```

Pour l'écriture, la classe `FileStream` expose deux méthodes `WriteByte` et `Write` permettant respectivement d'écrire une valeur binaire unique ou une série de valeurs provenant d'un tableau de type `byte` :

```
stream.WriteByte(50);

bytes = new byte[] { 15, 65, 98, 78, 126 };
stream.Write(bytes, 0, 5);
```

Lorsque les opérations sur le fichier sont terminées, il faut toujours le fermer sinon il reste verrouillé et aucun autre processus ne peut y accéder :

```
stream.Close();
```

La classe `FileStream` implémentant l'interface `IDisposable`, il convient également de libérer les ressources explicitement :

```
stream.Dispose();
```

3. Lire un fichier texte

a. Lire avec la classe File

La classe statique `File` expose des méthodes permettant de lire des données dans un fichier sous forme binaire ou texte sans avoir à instancier un flux.

La méthode `ReadAllText` prend en paramètre le chemin d'un fichier à lire. Elle ouvre ce fichier, en lit le contenu et le ferme avant de retourner un objet `string` du contenu du fichier :

```
string content = File.ReadAllText(@"C:\fichier.txt");
```

La méthode `ReadAllText` possède une surcharge permettant de spécifier l'encodage du contenu texte à l'aide d'un paramètre de type `Encoding` :

```
string content = File.ReadAllText(@"C:\fichier.txt",
Encoding.ASCII);
```

La méthode `ReadAllLines` fonctionne de la même manière que la méthode `ReadAllText` à l'exception que le contenu est retourné sous la forme d'un tableau de type `string`. Chaque élément correspond à une ligne du fichier source :

```
string[] lines = File.ReadAllLines(@"C:\fichier.txt");
```

La méthode `ReadLines` retourne un objet générique `IEnumerable` dont chaque élément représente une ligne du fichier source :

```
IEnumerable<string> lines = File.ReadLines(@"C:\fichier.txt");
```

La classe `File` expose une méthode permettant de lire des données binaires : la méthode `ReadAllBytes`. Elle prend en paramètre le chemin du fichier source et retourne un tableau de type `byte` représentant le contenu du fichier :

```
byte[] bytes = File.ReadAllBytes(@"C:\fichier.dat");
```

b. Lire avec la classe StreamReader

La classe `StreamReader` est utilisée pour lire des fichiers texte. La manière la plus simple d'instancier un nouvel objet `StreamReader` est de spécifier le chemin du fichier :

```
StreamReader stream = new StreamReader(@"C:\fichier.txt");
```

Si aucun encodage n'est spécifié, le constructeur examine les premiers bits pour déterminer l'encodage du fichier. Ces bits sont connus sous le nom de *byte code markers*. Ils sont tout simplement absents lorsque le fichier est encodé en ASCII pour des raisons de rétrocompatibilité avec les anciens systèmes ne gérant pas l'Unicode. Lorsque le fichier est encodé en Unicode, UTF7, UTF8 ou UTF32 les premiers bits de ce fichier sont positionnés de manière spécifique pour indiquer le format du fichier.

Des surcharges du constructeur de la classe `StreamReader` permettent de spécifier l'encodage avec un objet `Encoding` et si oui ou non l'encodage doit être déterminé à l'aide des *byte code markers* :

```
StreamReader stream = new StreamReader(@"C:\fichier.txt",
                                       Encoding.ASCII);
```

Lorsque le fichier est ouvert, vous avez plusieurs possibilités de lire les données. L'objet `StreamReader` conserve en mémoire la position du curseur permettant d'identifier les parties déjà lues du fichier.

La méthode `ReadLine` retourne un objet de type `string` correspondant à la ligne du curseur et positionne celui-ci à la ligne suivante :

```
string line = stream.ReadLine();
```

La méthode `Read` retourne la valeur sous forme d'entier du prochain caractère après le curseur ou -1 si la fin du fichier a été atteinte :

```
int nextChar = stream.Read();
```

La propriété booléenne `EndOfStream` est une autre manière de tester si la fin du fichier a été atteinte :

```
while (!stream.EndOfStream)
{
    string s = stream.ReadLine();
}
```

Une surcharge de la méthode `Read` permet de spécifier combien de caractères doivent être lu et de les insérer dans un tableau de type `char` :

```
char[] chars = new char[3];
int buffChar = stream.Read(chars, 0, 3);
```

La méthode `ReadToEnd` retourne un objet de type `string` contenant la totalité du fichier entre la position du curseur et la fin du fichier :

```
string all = stream.ReadToEnd();
```

Lorsque les opérations de lecture du fichier sont terminées, il faut toujours fermer le fichier sinon il reste verrouillé et aucun autre processus ne peut y accéder :

```
stream.Close();
```

La classe `StreamReader` implémentant l'interface `IDisposable`, il convient également de libérer les ressources explicitement :

```
stream.Dispose();
```

En prenant l'exemple d'un fichier dont le contenu est le suivant :

```
Contenu
du
fichier
```

Et les instructions suivantes :

```
StreamReader stream = new StreamReader(@"C:\fichier.txt",
                                       Encoding.ASCII);
string line = stream.ReadLine();
int nextChar = stream.Read();
char[] chars = new char[3];
int buffChar = stream.Read(chars, 0, 3);
string all = stream.ReadToEnd();
stream.Close();
stream.Dispose();
```

La valeur des variables à la fin de l'exécution des précédentes instructions sont les suivantes :

```
line = "Contenu"
nextChar = 100 soit le caractère 'd'
chars[0] = 'u'
chars[1] = '\r'
chars[2] = '\n'
all = "fichier"
```

4. Écrire dans un fichier texte

a. Écrire avec la classe File

La classe statique `File` expose des méthodes permettant d'écrire des données dans un fichier sous forme binaire ou texte sans avoir à instancier un flux.

La méthode `WriteAllText` prend en paramètres le chemin d'un fichier et le contenu à écrire. Elle crée un nouveau fichier, inscrit le contenu dans ce fichier, le sauvegarde puis le ferme. Si le fichier est déjà existant, il sera écrasé :

```
File.WriteAllText(@"C:\fichier.txt", "Contenu");
```

La méthode `WriteAllText` possède une surcharge permettant de spécifier l'encodage du contenu texte à l'aide d'un paramètre de type `Encoding` de l'espace de noms `System.Text` :

```
File.WriteAllText(@"C:\fichier.txt", "Contenu", Encoding.ASCII);
```

La méthode `WriteAllLines` fonctionne de la même manière que la méthode `WriteAllText` à l'exception que le contenu est passé soit dans un tableau de type `string`, soit dans une collection générique `IEnumerable`. Chaque élément sera écrit sur une ligne du fichier de destination :

```
string[] lines = { "Contenu", "du", "fichier" };
File.WriteAllLines(@"C:\fichier.txt", lines, Encoding.ASCII);

List<string> lines = new List<string>()
                        { "Contenu", "du", "fichier" };
File.WriteAllLines(@"C:\fichier.txt", lines, Encoding.ASCII);
```

La classe `File` expose une méthode permettant d'écrire des données binaires : la méthode `WriteAllBytes`. Elle prend en paramètres le chemin du fichier de destination ainsi qu'un tableau de type `byte` représentant le contenu du fichier :

```
byte[] B = { 1, 78, 96, 135 };
File.WriteAllBytes(@"C:\fichier.dat", B);
```

b. Écrire avec la classe StreamWriter

La classe `StreamWriter` est utilisée pour écrire dans des fichiers texte. La manière la plus simple d'instancier un nouvel objet `StreamWriter` est de spécifier le chemin du fichier :

```
StreamWriter stream = new StreamWriter(@"C:\fichier.txt");
```

Des surcharges du constructeur de la classe `StreamWriter` permettent de spécifier l'encodage avec un objet `Encoding` et si oui ou non le contenu devra être ajouté aux données existantes :

```
StreamWriter stream = new StreamWriter(@"C:\fichier.txt",
                                       false,
                                       Encoding.ASCII);
```

Les méthodes `Write` et `WriteLine` permettent d'écrire des données dans le fichier. Elles contiennent de nombreuses surcharges pour chacun des types de base comme les types `string`, `char`, `char[]`, `bool`, `int`, `long` ou `object` :

```
stream.WriteLine("Contenu");
stream.Write('d');
stream.Write(new char[] { 'u', '\r', '\n' });
stream.Write("fichier");
```

Lorsque les opérations d'écriture sur le fichier sont terminées, il faut toujours fermer le fichier sinon il reste verrouillé et aucun autre processus ne peut y accéder :

```
stream.Close();
```

La classe `StreamWriter` implémentant l'interface `IDisposable`, il convient également de libérer les ressources explicitement :

```
stream.Dispose();
```

Introduction

La sérialisation est le nom du processus qui permet de convertir un objet en un flux. Il est alors possible de l'enregistrer dans un fichier ou de le transmettre à une autre application. La récupération de ce flux et sa transformation en objet sont appelées la désérialisation.

Le Framework .NET met à disposition deux techniques :

1. La sérialisation binaire permet de créer une copie exacte de l'objet sérialisé. Toutes les propriétés publiques, privées, sa classe ou encore son assemblage sont transformés en un flux de données.

2. La sérialisation XML crée une représentation des propriétés et des champs publics uniquement. L'avantage est que cela permet une communication plus aisée entre des applications hétérogènes. La sérialisation SOAP qui est une variante est largement utilisée notamment pour les services Web.

La sérialisation binaire

La meilleure raison d'utiliser la sérialisation binaire est de rendre persistant l'état d'un objet afin de pouvoir le recréer à l'identique ultérieurement. Cet objet peut alors être stocké dans un fichier ou envoyé en tant que flux à travers le réseau. Un autre avantage est que la sérialisation binaire conserve l'état des membres publics et privés de l'objet.

A contrario, il n'est pas aisé d'utiliser les flux générés dans ou depuis une application tierce.

1. Les bases

Le pré requis pour rendre une classe sérialisable est de la marquer avec l'attribut `Serializable`. Tenter de sérialiser un objet qui ne serait pas marqué de cet attribut entraînerait la levée d'une exception de type `SerializationException` :

```
[Serializable]
public class ReplacedField
{
    public string Pattern;
    public string Field;
    public bool HasChanged;
}
```

L'exemple de code ci-dessous montre la méthode permettant à un objet de cette classe d'être sérialisé dans un fichier :

```
// Instanciation et initialisation de l'objet
ReplacedField O = new ReplacedField();
O.Field = "myField";
O.Pattern = "myPattern";
O.HasChanged = true;

// Création du formateur binaire
IFormatter formatter = new BinaryFormatter();

// Création du flux
Stream stream = new FileStream(@"C:\ReplacedField.bin",
FileMode.Create);

// Sérialisation de l'objet dans le flux
formatter.Serialize(stream, O);
stream.Close();
stream.Dispose();
```

La sérialisation utilise un formateur binaire du type `BinaryFormatter` de l'espace de noms `System.Runtime.Serialization.Formatters.Binary`. Pour effectuer la transformation, il suffit d'appeler la méthode `Serialize` de cet objet en passant en paramètres le flux dans lequel écrire les données et l'objet à sérialiser.

La désérialisation suit le même schéma et utilise également un formateur binaire pour lequel il faut appeler la méthode `Deserialize` en passant en paramètre le flux contenant l'objet. Elle donnera en retour un objet qu'il faudra convertir :

```
// Création du formateur binaire
IFormatter formatter = new BinaryFormatter();

// Création du flux
Stream stream = new FileStream(@"C:\ReplacedField.bin",
FileMode.Open);

// Désérialisation de l'objet à partir du flux
ReplacedField O = (ReplacedField)formatter.Deserialize(stream);
stream.Close();
stream.Dispose();
```

La sérialisation binaire conserve l'état des membres publics et privés. Pour qu'un membre ne soit pas sérialisé, il faut le marquer avec l'attribut `NonSerialized` :

```
[Serializable]
public class ReplacedField
{
    [NonSerialized]
    public string Pattern;
    public string Field;
    public bool HasChanged;
}
```

2. Contrôler la sérialisation

Il peut arriver que les processus de sérialisation et désérialisation doivent être contrôlés plus finement par exemple pour garantir la rétrocompatibilité d'un type d'une version à une autre.

a. Le contrôle par attribut

La première technique de contrôle utilise les attributs et indique au processus quelle méthode doit être invoquée :

- `OnSerializing` : appliqué à une méthode, cet attribut spécifie que celle-ci doit être appelée avant la sérialisation de l'objet.
- `OnSerialized` : appliqué à une méthode, cet attribut spécifie que celle-ci doit être appelée après la sérialisation de l'objet.
- `OnDeserializing` : appliqué à une méthode, cet attribut spécifie que celle-ci doit être appelée avant la désérialisation de l'objet.
- `OnDeserialized` : appliqué à une méthode, cet attribut spécifie que celle-ci doit être appelée après la désérialisation de l'objet.

```
[OnSerializing]
private void OnSerializingMethod(StreamingContext context)
{
    this.Field = "Valeur modifiée avant la sérialisation";
}

[OnSerialized]
private void OnSerializedMethod(StreamingContext context)
{
    this.Field = "Valeur modifiée après la sérialisation";
}

[OnDeserializing]
private void OnDeserializingMethod(StreamingContext context)
{
    this.Field = "Valeur modifiée avant la désérialisation";
}

[OnDeserialized]
private void OnDeserializedMethod(StreamingContext context)
{
    this.Field = "Valeur modifiée après la désérialisation";
}
```

Ces méthodes n'accèdent pas au flux de sérialisation mais permettent de modifier l'objet à un instant donné du processus. En cas d'héritage, les méthodes sont exécutées successivement de la base au type dérivé.

Ces attributs ne peuvent être utilisés qu'une fois sur une méthode du type. Si plusieurs méthodes sont marquées par le même attribut, le code pourra être compilé mais une exception du type `TypeLoadException` sera levée à l'exécution car le processus ne sait pas quelle méthode doit être appelée.

Les méthodes marquées par ces attributs doivent accepter un paramètre du type `StreamingContext` de l'espace de noms `System.Runtime.Serialization` qui fournit des informations supplémentaires sur la source et la destination du

flux transmis.

Il est également possible de définir l'attribut `OptionalField` qui, appliqué à un membre, permet de spécifier au formateur que celui-ci peut être absent du flux et de ne pas lever d'exception :

```
[OptionalField]
public int NewField;
```

Cet attribut est utile pour garantir l'évolution d'un type. Lorsque le formateur désérialisera le type d'une ancienne version ne comportant pas ce nouveau champ, aucune exception ne sera levée et les autres membres seront traités correctement.

La bonne pratique consiste à utiliser la propriété `VersionAdded` de l'attribut `OptionalField` pour garantir la compatibilité avec les futurs moteurs de sérialisation. Cette valeur doit être incrémentée de 1 pour chaque modification de version du type en commençant par 2 :

```
// Version 1
[Serializable]
public class ReplacedField
{
    public string Pattern;
}
// Version 2
[Serializable]
public class ReplacedField
{
    public string Pattern;
    [OptionalField(VersionAdded = 2)]
    public string Field;
}
// Version 3
[Serializable]
public class ReplacedField
{
    public string Pattern;
    [OptionalField(VersionAdded = 2)]
    public string Field;
    [OptionalField(VersionAdded = 3)]
    public bool HasChanged;
    [OptionalField(VersionAdded = 3)]
    public string NewField;
}
```

Pour assurer la rétrocompatibilité de la sérialisation d'un type, certaines règles doivent être suivies :

- Ne jamais supprimer un champ sérialisé.
- Ne jamais appliquer l'attribut `NonSerialized` à un membre s'il n'était pas appliqué dans sa version antérieure.
- Ne jamais modifier le nom ou le type d'un membre sérialisé.
- Un nouveau membre ou un membre qui était marqué par l'attribut `NonSerialized` et qui ne l'est plus, doit être marqué par l'attribut `OptionalField`.

b. Le contrôle par interface

Une seconde technique de contrôle du processus de sérialisation est l'implémentation de l'interface `ISerializable` au type. L'implémentation de cette interface implique l'implémentation de la méthode `GetObjectData` et d'un constructeur spécifique pour la désérialisation :

```
[Serializable]
public class ReplacedField : ISerializable
{
    public string Pattern;
    public string Field;
    public bool HasChanged;
```

```
    public ReplacedField() { }

    internal ReplacedField(SerializationInfo info,
StreamingContext context)
    {
        Pattern = info.GetString("Pattern");
        Field = info.GetString("Field");
        HasChanged = info.GetBoolean("HasChanged");
    }

    public virtual void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("Pattern", Pattern);
        info.AddValue("Field", Field);
        info.AddValue("HasChanged", HasChanged);
    }
}
```

L'absence de la méthode `GetObjectData` lèvera une erreur au moment de la compilation et l'empêchera, mais l'absence du constructeur spécifique n'empêchera pas le code de compiler et une erreur du type `SerializationException` sera levée au moment de l'exécution.

Lors de la sérialisation, le processus appel la méthode `GetObjectData` qui prend en paramètres un objet `SerializationInfo` et un objet `StreamingContext`. Le corps de la méthode doit remplir l'objet `SerializationInfo` avec les membres souhaités de l'objet sous forme de paires de clés et de valeurs. Les noms des clés sont libres mais doivent être utilisés à l'identique lors de la désérialisation :

- Ajout d'une valeur aux informations de sérialisation dans la méthode `GetObjectData` :

```
info.AddValue("Pattern", Pattern);
```

- Récupération de cette valeur au moment de la désérialisation dans le constructeur spécifique :

```
Pattern = info.GetString("Pattern");
```

Le processus de désérialisation se sert du constructeur spécifique pour passer l'objet `SerializationInfo` contenant les informations sur les membres. Le modificateur d'accès du constructeur n'est pas pris en compte pour l'appel de la méthode donc, sauf contrainte, ce constructeur peut être marqué avec le modificateur `internal` pour éviter toute utilisation par un code tiers.

Pour restaurer l'objet dans son état, il suffit d'affecter aux membres les valeurs définies lors de la sérialisation dans l'objet `SerializationInfo` :

```
internal ReplacedField(SerializationInfo info, StreamingContext
context)
{
    Pattern = info.GetString("Pattern");
    Field = info.GetString("Field");
    HasChanged = info.GetBoolean("HasChanged");
}
```

Lors de la création d'une classe dérivant d'une autre qui implémente l'interface `ISerializable`, la nouvelle classe n'a pas besoin d'indiquer explicitement l'interface, mais elle doit implémenter la méthode `GetObjectData` et le constructeur spécifique :

```
public class ReplacedFieldChild : ReplacedField
{
    public string NewField;

    public ReplacedFieldChild() : base() { }

    internal ReplacedFieldChild(SerializationInfo info,
StreamingContext context)
        : base(info, context)
    {
```

```
        NewField = info.GetString("NewField");
    }

    public override void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        base.GetObjectData(info, context);
        info.AddValue("NewField", NewField);
    }
}
```

Lors de la sérialisation et de la désérialisation dans la classe héritée, il faut faire appel explicitement aux méthodes de la classe de base sans quoi, elles ne seraient jamais appelées et le flux de données sérialisés ou l'objet obtenu après désérialisation serait incomplet :

```
private ReplacedFieldChild(SerializationInfo info,
StreamingContext context)
        : base(info, context)
```

ou

```
base.GetObjectData(info, context);
```

Pour les cas où des objets enfants doivent être instanciés après que la désérialisation soit finie, il est possible d'implémenter l'interface `IDeserializationCallback` et la méthode `IDeserializationCallback.OnDeserialization` afin de réaliser des tâches supplémentaires :

```
public class ReplacedFieldChild : ReplacedField,
IDeserializationCallback
{
  // ...

  void IDeserializationCallback.OnDeserialization(Object sender)
  {
    NewField = "Valeur modifiée par la méthode de Callback";
  }
}
```

3. La sérialisation XML

Le processus de sérialisation XML ne s'occupe que des champs publics, sans informations sur les types, les champs privés ou les propriétés en lecture seule. Le principal avantage de la sérialisation XML est sa flexibilité au niveau du flux créé. Il permet de transporter facilement des données d'une application à une autre.

4. Les bases

Les processus de sérialisation et désérialisation XML suivent les mêmes principes que la sérialisation binaire à l'exception près que la classe n'a pas besoin d'être marquée par l'attribut `Serializable` :

```
public class ReplacedField
{
    public string Pattern;
    public string Field;
    public bool HasChanged;
}
```

La classe `XmlSerializer` de l'espace de noms `System.Xml.Serialization` est utilisée pour effectuer les transformations :

```
// Instanciation et initialisation de l'objet
ReplacedField O = new ReplacedField();
O.Field = "myField";
O.Pattern = "myPattern";
O.HasChanged = true;
```

```
// Création du sérialiseur XML
XmlSerializer serializer = new
XmlSerializer(typeof(ReplacedField));

// Création du flux
Stream stream = new FileStream(@"C:\ReplacedField.xml",
FileMode.Create);

// Sérialisation de l'objet dans le flux
serializer.Serialize(stream, O);
stream.Close();
```

L'objet `XmlSerializer` prend en paramètre le type de l'objet à sérialiser. L'appel de la méthode `Serialize` de l'objet, en passant en paramètres le flux dans lequel écrire les données et l'objet à sérialiser, effectue la transformation.

Le résultat de la sérialisation précédente est le suivant :

```
<?xml version="1.0"?>
<ReplacedField xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Pattern>myPattern</Pattern>
  <Field>myField</Field>
  <HasChanged>true</HasChanged>
</ReplacedField>
```

L'élément racine du document possède le nom de la classe sérialisée et les nœuds enfants ont le nom de leur membre correspondant.

Comme pour la désérialisation binaire, l'appel de la méthode `Deserialize` de l'objet `XmlSerializer` en passant en paramètre le flux contenant l'objet donnera en retour un objet à convertir :

```
// Création du sérialiseur XML
XmlSerializer serializer = new
XmlSerializer(typeof(ReplacedField));

// Création du flux
Stream stream = new FileStream(@"C:\ReplacedField.xml",
FileMode.Open);

// Désérialisation de l'objet à partir du flux
ReplacedField O = (ReplacedField)serializer.Deserialize(stream);
stream.Close();
```

Pour illustrer cette section, créez une nouvelle classe générique et statique nommée `Serializer` dans le dossier **Library** du projet :

```
public static class Serializer<T> where T : class
{
}
```

Ajoutez une première méthode `Serialize` permettant de sérialiser un objet dans un fichier :

```
public static void Serialize(T o, string file)
{
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(T));
    Stream stream = new FileStream(file, FileMode.Create);
    xmlSerializer.Serialize(stream, o);
    stream.Close();
    stream.Dispose();
}
```

Ajoutez une seconde méthode `Deserialize` permettant de désérialiser une classe à partir d'un fichier :

```
public static T Deserialize(string file)
{
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(T));
    Stream stream = new FileStream(file, FileMode.Open);
    T result = xmlSerializer.Deserialize(stream) as T;
    stream.Close();
    stream.Dispose();
```

```
    return result;
}
```

Vous pouvez désormais ajouter les instructions de sauvegarde dans la méthode `Save` de la classe `Project` :

```
Serializer<Project>.Serialize(this,
                              System.IO.Path.Combine(this.Path,
                                                     this.Filename));
```

Modifiez la méthode `Load` de la classe `Project` de la manière suivante :

```
public static Project Load(string FilePath)
{
    Project result = Serializer<Project>.Deserialize(FilePath);
    result.Path = System.IO.Path.GetDirectoryName(FilePath);
    result.Filename = System.IO.Path.GetFileName(FilePath);
    result.HasChanged = false;

    result.ProjectSettings.Changed += new
EventHandler<ChangedEventArgs>(result.ChildChanged);
    result.MailServerSettings.Changed += new
EventHandler<ChangedEventArgs>(result.ChildChanged);
    result.MailProperties.Changed += new
EventHandler<ChangedEventArgs>(result.ChildChanged);

    return result;
}
```

Terminez par modifier la méthode `chargerToolStripMenuItem_Click` du formulaire `Main` :

```
Program.Project = Library.Project.Load(this.ofdProject.FileName);
```

Lancez l'application (**F5**) pour tester la sauvegarde et le chargement de données.

5. Contrôler la sérialisation

Il est souvent utile de pouvoir contrôler le processus de sérialisation pour se conformer aux spécifications d'une application tierce avec laquelle il faut partager des données et dont le schéma de données est rigide. L'utilisation d'attributs permet de gérer les différents cas.

Comme vu précédemment, un élément XML porte le nom de la classe ou du membre dont il est issu. La modification de ce comportement pour une classe peut être obtenue grâce à l'attribut `XmlRoot` en définissant sa propriété `ElementName` :

```
[XmlRoot(ElementName = "RootElement")]
public class ReplacedField
```

La sérialisation donnera le résultat suivant :

```
<?xml version="1.0"?>
<RootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ...
</RootElement>
```

Il est possible de préciser d'autres propriétés comme `Namespace` pour affiner le flux généré :

```
[XmlRoot(ElementName = "RootElement", Namespace = "myNamespace")]
public class ReplacedField
```

La sérialisation donnera le résultat suivant :

```
<?xml version="1.0"?>
<RootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="myNamespace">
  ...
```

```
</RootElement>
```

L'attribut `XmlElement` a le même effet sur les membres de la classe :

```
[XmlElement(ElementName = "PatternElement")]
public string Pattern;
```

La sérialisation donnera le résultat suivant :

```
<?xml version="1.0"?>
<RootElement ...>
  <PatternElement>myPattern</PatternElement>
  ...
</RootElement>
```

L'attribut `XmlIgnore` permet de ne pas sérialiser le membre marqué. Il n'apparaîtra donc pas dans le flux généré et ne sera pas pris en compte lors de la désérialisation.

Ajoutez l'attribut `XmlIgnore` sur les propriétés `HasChanged` des classes du dossier **Library** ainsi que sur les propriétés `Filename` et `Path` de la classe `Project`.

Parmi les attributs les plus communs, `XmlAttribute` spécifie que le membre doit être sérialisé comme un attribut XML :

```
 [XmlAttribute(AttributeName = "PatternElement")]
public string Pattern;
```

La sérialisation donnera le résultat suivant :

```
<?xml version="1.0"?>
<RootElement ... PatternElement="myPattern">
  ...
</RootElement>
```

6. La sérialisation XML SOAP

SOAP (*Simple Object Access Protocole*) est un format spécifique de fichier XML mis en place pour faciliter la description des objets complexes entre systèmes d'informations. Ce protocole est principalement utilisé par les services Web.

Le fonctionnement de la sérialisation SOAP est identique à celui de la sérialisation binaire en utilisant le type `SoapFormatter` de l'espace de noms `System.Runtime.Serialization.Formatters.Soap`. Cette librairie n'est pas incluse par défaut dans un projet d'application Windows, il faut donc ajouter une référence au projet vers la librairie .NET nommée **System.Runtime.Serialization.Formatters.Soap** :

```
// Instanciation et initialisation de l'objet
Version O = new Version(1,0);
// Création du formateur SOAP
SoapFormatter formatter = new SoapFormatter();
// Création du flux
Stream stream = new FileStream(@"C:\Soap.xml", FileMode.Create);
// Sérialisation de l'objet dans le flux
formatter.Serialize(stream, O);
stream.Close();
```

L'exemple de sérialisation ci-dessus produit 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:Version id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/ns/System">
      <_Major>1</_Major>
      <_Minor>0</_Minor>
```

```
      <_Build>-1</_Build>
      <_Revision>-1</_Revision>
    </a1:Version>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Comme pour la sérialisation XML, les attributs permettent de modifier le comportement du processus avec notamment `SoapAttribute`, `SoapElement` ou `SoapIgnore` qui ont le même effet que les attributs de sérialisation XML.

Introduction

Les expressions régulières ont pour but de rechercher des chaînes de caractères dans une autre chaîne de caractères à l'aide de modèles déterminés. Ces modèles sont définis sous forme d'objets de type `string` en une syntaxe précise comme pour un langage spécifique. L'espace de noms `System.Text.RegularExpressions` expose les classes utilisées pour travailler avec les expressions régulières comme `Regex`, `Match` ou `Group`.

Les expressions régulières permettent de réaliser différentes opérations sur les objets de type `string`. Vous pouvez par exemple identifier les mots en doublon, changer la casse, séparer les différents éléments d'une URL ou vérifiez que le format correspond bien à un modèle.

L'utilisation des expressions régulières se fait essentiellement avec la classe `Regex` qui contient des méthodes statiques et qui peut aussi être instanciée. Une expression régulière ressemble de prêt à un objet `string`. La différence est qu'il contient des séquences d'échappement et des séries de caractères ayant un but spécifique.

Les exemples de ce chapitre sont disponibles dans les sources sous le projet RegularExpressions de la solution. La constante de type string utilisée pour les exemples est la suivante :

```
const string Original =
"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mio
enim sapien, dignissim in eleifend mcc, molestie scr amet sapien
Proin in ligula 022-999, eget eleifend urna. Nunc sollicitudin
elementum agr, ddu mattis arcu auctor mfi? Mauris vulputate
condimentum venenatis. Curabitur semper faucibus arcu mpe
sagittis.";
```

Une première expression régulière

La chaîne `Original` est considérée comme étant la chaîne d'entrée. C'est sur celle-ci que les expressions régulières seront testées. Pour vérifier l'existence d'une série d'une chaîne dans celle-ci, il faudra exécuter les instructions suivantes :

```
string pattern = "mio";
MatchCollection matches = Regex.Matches(Original,
                                        pattern,
                                        RegexOptions.IgnoreCase);
foreach (Match match in matches)
{
    Console.WriteLine(match.Index);
}
```

Cet exemple définit un modèle, la chaîne de caractères `mio`. Ensuite, on fait appel à la méthode statique `Matches` de la classe `Regex` en passant comme arguments, la chaîne originale, le schéma à retrouver et en spécifiant l'option `RegexOptions.IgnoreCase` permettant de ne pas prendre en compte la casse lors de la recherche. La méthode `Matches` retourne un objet de type `MatchCollection`, c'est à dire une collection d'objets `Match` représentant chacun un résultat. L'exemple se termine sur une itération de la collection de résultats pour en afficher l'index correspondant à la position du premier caractère du modèle dans la chaîne originale. En lançant l'application (**F5**), le résultat affiché est `57`, cela signifie que nous avons un résultat dans la chaîne originale et qu'il se situe à l'index `57`.

La méthode statique `Match` de la classe `Regex` permet de retourner le premier résultat valide correspondant au modèle.

Les options de recherche

L'énumération `RegexOptions` est marquée avec l'attribut `Flags`. Cela signifie que ses valeurs peuvent être combinées avec des opérateurs de bits :

```
RegexOptions searchOptions = RegexOptions.IgnoreCase |
                             RegexOptions.Multiline;
```

La liste suivante présente les différentes valeurs de l'énumération `RegexOptions` :

- `None` : spécifie qu'aucune option n'est définie. Si cette valeur est combinée avec une autre, elle sera ignorée.
- `IgnoreCase` : spécifie que la casse est ignorée lors de la recherche.
- `Multiline` : modifie la signification des caractères `^` et `$` de manière à ce qu'ils soient appliqués sur chaque ligne de la chaîne originale et non pas au début et à la fin de la chaîne entière.
- `ExplicitCapture` : modifie la manière dont les résultats sont collectés en étant sûr que les captures valides sont celles qui sont explicitement nommées.
- `Compiled` : spécifie que l'expression régulière est compilée dans un assemblage. L'exécution est plus rapide, mais le temps de démarrage est augmenté.
- `Singleline` : modifie le sens du caractère . de manière à ce qu'il corresponde à n'importe quel caractère.
- `IgnorePatternWhitespace` : supprime les espaces non échappés du modèle et active les commentaires marqués avec le caractère `#`.
- `RightToLeft` : modifie le sens de la recherche qui sera effectuée de droite à gauche au lieu de gauche à droite par défaut.
- `ECMAScript` : cette option active un comportement conforme ECMAScript pour l'expression. Il faut combiner cette option avec les valeurs `IgnoreCase`, `Multiline` et `Compiled` sans quoi une exception serait levée.
- `CultureInvariant` : spécifie que la culture de la chaîne est ignorée.

Les caractères d'échappement

Les expressions régulières contiennent des caractères permettant de spécifier des parties de modèles. Ils doivent être précédés par un antislash \ de manière à être considérés comme un caractère réel. Par exemple, le caractère ? signifie que le caractère précédent peut apparaître zéro ou une fois. Pour faire une recherche sur le point d'interrogation lui-même, il faut le préfixer d'un antislash :

```
Console.WriteLine(Regex.Match(Original, @"mfi?"));
Console.WriteLine(Regex.Match(Original, @"mfi\?"));
```

Le résultat affiché est le suivant :

```
mfi
mfi?
```

Le résultat n'est pas identique suivant que le caractère ? est échappé ou non. Dans la première instruction, l'expression régulière recherche une suite de caractères pouvant être mf ou mfi alors que pour la seconde, la recherche porte sur la suite exacte mfi?.

Les méthodes statiques Escape et Unescape de la classe Regex permettent de transformer les caractères dans leur équivalent échappé :

```
Console.WriteLine(Regex.Escape(@"?"));       // \?
Console.WriteLine(Regex.Unescape(@"\?"));    // ?
```

Les ensembles

Les ensembles de caractères permettent de spécifier une série de caractères possible dans le modèle. Pour avoir une correspondance précise avec un ensemble de caractères, il faut les indiquer entre crochets :

```
Console.WriteLine(Regex.Match(Original, @"sc[Rr]"));    // scr
```

Le caractère ^ positionné en premier après le crochet d'ouverture permet de spécifier l'inverse :

```
Console.WriteLine(Regex.Match(Original, @"sc[^Rr]"));    // sci
```

Il possible de spécifier une suite de caractères en séparant le premier du dernier par un tiret. Cela évite de devoir spécifier tous les caractères un par un :

```
Console.WriteLine(Regex.Match(Original, @"sc[n-z]"));    // scr
```

`\d` représente un chiffre. Son contraire est `\D` qui signifie tout caractère sauf un chiffre :

```
Console.WriteLine(Regex.Match(Original, @"\d\D\d"));    // 2-9
```

`\w` est le diminutif de l'expression `[a-zA-Z0-9_]`. Il permet de rechercher tout caractère pouvant composer un mot :

```
Console.WriteLine(Regex.Match(Original, @"\w"));            // L
Console.WriteLine(Regex.Match(Original, @"[a-zA-Z0-9_]"));  // L
```

`\s` permet de spécifier un espace dans le modèle :

```
Console.WriteLine(Regex.Match(Original, @"Lorem\sipsum"));
// Lorem ipsum
```

Le caractère . correspond à tout caractère à l'exception de `\n` :

```
Console.WriteLine(Regex.Match(Original, @"m.e"));        // mpe
```

`\p` suivi d'une catégorie entre accolades permet de spécifier une catégorie. Les catégories sont les suivantes :

- `L` pour les lettres.
- `Ll` pour les lettres minuscules.
- `Lu` pour les lettres majuscules.
- `N` pour les nombres.
- `P` pour la ponctuation.
- `S` pour les symboles.
- `Z` pour les séparateurs

```
Console.WriteLine(Regex.Match(Original, @"d\p{L}u"));    // ddu
```

Les groupes

Il est parfois utile de pouvoir séparer une expression régulière en une série de plusieurs sous expressions appelées des groupes. Pour former un groupe, l'expression doit être entourée de parenthèses :

```
Match result = Regex.Match(Original, @"(\d\d\d)-(\d\d\d)");
Console.WriteLine(result);                          // 022-999
Console.WriteLine(result.Groups[0]);                // 022-999
Console.WriteLine(result.Groups[1]);                // 022
Console.WriteLine(result.Groups[2]);                // 999
```

Dans cet exemple, vous pouvez remarquer que le résultat correspond au résultat complet de l'expression régulière tout comme le premier élément de la collection de groupes qui a l'index 0. Les groupes suivants sont ensuite formés par les groupes déterminés par les parenthèses dans leur ordre d'apparition dans l'expression régulière.

En plus de pouvoir séparer les résultats, les groupes peuvent également être utilisés au sein de l'expression régulière. Les groupes sont indexés séquentiellement en commençant par 1. Ainsi, vous pouvez faire référence à un groupe dans l'expression régulière avec l'index du groupe échappé :

```
foreach (Match match in Regex.Matches(Original, @"(\w)\1"))
{
    Console.WriteLine(match.Value);
}
```

Cet exemple fait une recherche sur les chaînes de deux caractères dont le second est égal au premier. Les résultats sont les suivants :

```
ss
cc
22
99
ll
dd
tt
tt
```

Les groupes capturés peuvent être nommés. La syntaxe est la suivante pour la déclaration :

```
(?'NomDuGroupe'Expression) ou (?<NomDuGroupe>Expression)
```

Pour utiliser un groupe nommé, utilisez la syntaxe suivante :

```
\k' NomDuGroupe ' ou \k< NomDuGroupe >
```

Voici un exemple identique au précédent en utilisant un groupe nommé :

```
foreach (Match match in Regex.Matches(Original,
                                  @"(?'lettre'\w)\k<lettre>"))
{
    Console.WriteLine(match.Value);
}
```

Les ancres

Les caractères ^ et $ permettent de spécifier une position particulière. Par défaut ^ correspond au début de la chaîne et $ correspond à la fin de celle-ci. Ainsi pour faire une recherche sur le début de la chaîne on utilisera une expression régulière de la forme suivante :

```
Console.WriteLine(Regex.Match(Original, @"^\w"));        // L
```

Suivant le contexte d'utilisation de ces caractères, ils ont une signification différente. ^ peut être une ancre de début de chaîne ou un caractère de négation tandis que $ peut signifier la fin de la chaîne ou précéder un identifiant de groupe en cas de remplacement.

Si l'option `RegexOptions.Multiline` est spécifiée, ^ correspondra au début de la chaîne ou de la ligne tandis que $ correspondra à la fin de la chaîne ou de la ligne.

Les quantifieurs

Les quantifieurs permettent de spécifier le nombre d'occurrences d'une correspondance. Le tableau suivant énumère les quantifieurs disponibles :

Quantifieur	Description
`*`	Zéro ou plusieurs correspondances.
`+`	Une ou plusieurs correspondances.
`?`	Zéro ou une correspondance.
`{x}`	Exactement x correspondances.
`{x,}`	Au moins x correspondances.
`{x,y}`	Entre x et y correspondances.

Un quantifieur s'applique au caractère ou groupe le précédent. Pour exemple, reprenons la précédente expression régulière : `(\d\d\d)-(\d\d\d)`. Elle recherche deux séries de trois chiffres séparées par un tiret. Elle peut donc être convertie en `(\d{3})-(\d{3})` :

```
Console.WriteLine(Regex.Match(Original, @"(\d{3})-(\d{3})"));
// 022-999
```

Introduction

La programmation multi-thread implique la répartition des tâches (thread) d'une même application de manière à ce qu'elles soient réalisées indépendamment les unes des autres afin d'exploiter au mieux le temps du processeur. Les tâches ne se réalisent pas réellement en parallèle. Le processeur affecte un temps de traitement à chaque tâche en fonction de leur importance et fait lui même le basculement de l'une à l'autre. Ce basculement de contexte implique que le processeur mémorise la pile de la tâche en cours avant de restaurer celle de la tâche à laquelle il redonne la main.

Les exemples de ce chapitre sont disponibles dans les sources sous le projet MultiThreading de la solution.

La classe Thread

L'espace de noms `System.Threading` héberge les classes permettant de créer et contrôler les tâches, notamment avec la classe `Thread`. Les threads doivent être utilisés lorsque, par exemple, une application doit gérer plusieurs tâches indépendantes comme gérer une interface utilisateur et réaliser un traitement de données. L'application aura de meilleures performances si ces deux tâches se déroulent dans des threads spécifiques.

1. Créer un thread

Le délégué `ThreadStart` est employé pour créer un nouveau thread. Son constructeur prend comme argument la signature de la méthode qui sera exécutée dans ce nouveau thread :

```
ThreadStart newThread = new ThreadStart(OtherThread);
```

Le délégué est ensuite passé en paramètre du constructeur de l'objet `Thread` :

```
Thread thread = new Thread(newThread);
```

Ensuite on appelle la méthode `Start` de l'objet `Thread` pour lancer l'appel à la méthode du délégué `ThreadStart` :

```
thread.Start();
```

L'exemple complet sous forme d'application console est le suivant :

```
static void Main(string[] args)
{
    Console.WriteLine("Lancement du thread principal");
    ThreadStart newThread = new ThreadStart(OtherThread);
    Thread thread = new Thread(newThread);
    thread.Start();
    Console.WriteLine("Thread principal terminé");
}
public static void OtherThread()
{
    Console.WriteLine("Lancement du thread secondaire");

    Console.WriteLine("Thread secondaire terminé");
}
```

Lorsque vous lancez cette application (**F5**), la console affiche le résultat suivant :

```
Lancement du thread principal
Thread principal terminé
Lancement du thread secondaire
Thread secondaire terminé
```

Vous pouvez remarquer que la méthode `Main` s'est complètement exécutée avant même que la méthode `OtherThread` n'ait débuté. Cela prouve que les deux méthodes se sont exécutées de manière asynchrone et donc dans des threads différents.

La propriété statique `CurrentThread` de la classe `Thread` vous permet d'accéder au thread en cours dans le contexte d'exécution. Dans l'exemple précédent, cela signifie que la propriété `CurrentThread` n'aura pas la même valeur suivant la méthode (`Main` ou `OtherThread`) dans laquelle elle sera appelée.

2. Suspendre ou annuler un thread

Le contrôle des threads est assez fin pour pouvoir les suspendre pendant un laps de temps déterminé. Il suffit d'appeler la méthode statique `Sleep` de la classe `Thread` :

```
Thread.Sleep(5000);
```

Cette méthode ne fonctionne qu'avec le thread courant. Si vous appelez la méthode `Sleep` en lui passant la valeur zéro comme paramètre, cela n'aura aucun effet.

L'annulation d'un thread se fait en appelant la méthode `Abort` de la classe `Thread`. Cette opération peut être exécutée sur le thread courant ou sur un autre thread :

```
static void Main(string[] args)
{
    Thread thread = new Thread(new ThreadStart(OtherThread));
    thread.Start();
    Console.WriteLine("Annulation du thread depuis un autre");
    thread.Abort();

    thread = new Thread(new ThreadStart(AbortThread));
    thread.Start();
}
public static void AbortThread()
{
    Console.WriteLine("Annulation du thread depuis
Thread.CurrentThread.Abort");
    Thread.CurrentThread.Abort();
}
```

3. Échanger des données avec un thread

Il existe deux manières de passer des données à un thread. La première consiste à utiliser un délégué du type `ParameterizedThreadStart` lors de l'instanciation du nouveau thread. Ce délégué accepte les méthodes dont la signature accepte en paramètre un argument de type `object`. La valeur de cet objet est ensuite transmise au thread lors de l'appel de la méthode `Start` :

```
static void Main(string[] args)
{
    Thread thread = new Thread(
              new ParameterizedThreadStart(ParameterizedThread));
    thread.Start(10);
}
public static void ParameterizedThread(object o)
{
    Console.WriteLine("Valeur passée au thread: {0}", o);
}
```

Le résultat de cet exemple est le suivant :

```
Valeur passée au thread: 10
```

Cette manière est simple à mettre en œuvre mais possède un inconvénient qui est que le type du paramètre n'est pas garanti car tout type d'objet est accepté par la méthode `Start`.

La seconde manière de passer des données à un thread est d'encapsuler la méthode du thread et les données dans une classe. L'objet instancié sera initialisé avec les valeurs désirées et le nouveau thread sera lancé à partir de la méthode de cet objet. Ainsi, le nouveau thread a accès aux données assignées par un autre thread :

```
class Program
{
    static void Main(string[] args)
    {
        CustomThread cThread = new CustomThread()
        {
            intValue = 10,
            stringValue = "Ma valeur"
        };
        Thread thread = new Thread(
                        new ThreadStart(cThread.ThreadMethod));
        thread.Start();
    }
}
public class CustomThread
{
    public int intValue { get; set; }
```

```
    public string stringValue { get; set; }

    public void ThreadMethod()
    {
        Console.WriteLine("Valeurs passée au thread:");
        Console.WriteLine("intValue = {0}", intValue);
        Console.WriteLine("stringValue = {0}", stringValue);
    }
}
```

Tout comme il est intéressant de pouvoir passer des valeurs à un thread, il est aussi juste de vouloir récupérer des données en retour de l'exécution d'un thread. La technique consiste à reprendre l'encapsulation de la méthode du thread dans une classe à laquelle on ajoute un délégué personnalisé. À la fin de l'exécution de la méthode, la méthode de retour est appelée via le délégué :

```
public delegate void CustomCallback(string returnValue);

class Program
{
    static void Main(string[] args)
    {
        CustomThread cThread = new CustomThread()
        {
            intValue = 10,
            stringValue = "Ma valeur"
            Callback = new CustomCallback(DisplayCallback)
        };
        Thread thread = new Thread(
                        new ThreadStart(cThread.ThreadMethod));
        thread.Start();
    }
    public static void DisplayCallback(string s)
    {
         Console.WriteLine("Valeur retournée: {0}", s);
    }
}
public class CustomThread
{
    public int intValue { get; set; }
    public string stringValue { get; set; }
    public CustomCallback Callback { get; set; }

    public void ThreadMethod()
    {
        if (Callback != null)
            Callback(string.Format("{0}: {1}",
                                   stringValue,
                                   intValue));
    }
}
```

Lors de l'instanciation de l'objet `CustomThread`, le délégué `Callback` est initialisé avec la méthode `DisplayCallback`. À la fin de la méthode `ThreadMethod`, la méthode associée au délégué de l'objet est appelée. Le résultat produit est le suivant :

```
Valeur retournée par le thread: Ma valeur: 10
```

4. Verrouiller un thread

Un thread peut être verrouillé de manière à ce qu'il puisse finir sans qu'une nouvelle instance de la méthode ne soit exécutée. Le mot clé `lock` encadre la portion de code qui doit être verrouillée. Si une nouvelle instance de la méthode doit être exécutée dans un nouveau thread, et que l'exécution du premier appel se trouve toujours verrouillée, le second appel sera placé en file d'attente et sera automatiquement exécuté dès que le déverrouillage aura eu lieu :

```
static void Main(string[] args)
{
    CustomThread cThread = new CustomThread();
```

```
    Thread thread1 = new Thread(
                      new ThreadStart(cThread.LockedThread));
    thread1.Start();
    Thread thread2 = new Thread(
                      new ThreadStart(cThread.LockedThread));
    thread2.Start();
}
public class CustomThread
{
    public void LockedThread()
    {
        lock (this)
        {
            Console.WriteLine("Le thread est verrouillé");
            Thread.Sleep(2000);
            Console.WriteLine("Le thread est déverrouillé");
        }
    }
}
```

5. Priorité des threads

Lors du basculement de contexte, le processeur choisit le prochain thread à exécuter en fonction de leurs priorités. Par défaut, la priorité d'un thread est `Normal`. La propriété `Priority` de la classe `Thread` permet de modifier cette priorité relative avec l'une des valeurs de l'énumération `ThreadPriority` : `Highest`, `AboveNormal`, `Normal`, `BelowNormal` et `Lowest` :

```
static void Main(string[] args)
{
    Console.WriteLine("Lancement de threads sans priorité");
    Thread thread1 = new Thread(new ThreadStart(Counter1));
    Thread thread2 = new Thread(new ThreadStart(Counter2));
    thread1.Start();
    thread2.Start();

    Console.ReadLine();

    Console.WriteLine("Lancement de threads avec priorité");
    thread1 = new Thread(new ThreadStart(Counter1));
    thread2 = new Thread(new ThreadStart(Counter2));
    thread1.Priority = ThreadPriority.Lowest;
    thread2.Priority = ThreadPriority.Highest;
    thread1.Start();
    thread2.Start();
}
public static void Counter1()
{
    for (int i = 1; i <= 10; i++)
    {
        Console.Write("{0} ", i);
    }
}
public static void Counter2()
{
    for (int i = 11; i <= 20; i++)
    {
        Console.Write("{0} ", i);
    }
}
```

La sortie produite par cet exemple est la suivante :

```
Lancement de threads sans priorité
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Lancement de threads avec priorité
11 12 13 14 15 16 17 18 19 20 1 2 3 4 5 6 7 8 9 10
```

Sans affecter de priorité aux threads, ils sont exécutés en se partageant également le temps processeur. Pour bien voir l'alternance entre les deux threads, il faut que les méthodes soient beaucoup plus longues. En modifiant la priorité des threads, l'un en priorité basse et le second en priorité haute, on observe que le thread de priorité haute s'est exécuté en priorité sur le thread de priorité basse.

Le composant BackgroundWorker

Le composant BackgroundWorker est utilisé dans le formulaire Send de la solution de démonstration.

La boîte à outils de Visual Studio fournit le composant `BackgroundWorker`. Il est très utile pour les applications Windows qui doivent à la fois gérer une interface utilisateur et son rafraîchissement mais aussi effectuer des opérations lourdes.

Comme indiqué par l'interface, le composant `BackgroundWorker` permet d'exécuter une opération dans un thread séparé. Il suffit d'instancier un nouvel objet `BackgroundWorker` soit via le concepteur de vue soit dans le code et de lui indiquer la méthode à exécuter dans un thread séparé :

```
BackgroundWorker bw = new BackgroundWorker();
bw.DoWork += new DoWorkEventHandler(bw_DoWork);
```

Lorsque vous souhaitez lancer l'exécution des tâches en arrière-plan, appelez la méthode `RunWorkerAsync` de l'objet `BackgroundWorker` :

```
bw.RunWorkerAsync();
```

Le composant a été simplifié au maximum pour faciliter son utilisation. Il contient deux propriétés `WorkerReportsProgress` et `WorkerSupportsCancellation` permettant respectivement de spécifier si le composant signale sa progression aux objets abonnés à son évènement `ProgressChanged` et si l'exécution de la tâche de fond peut être annulée via la méthode `CancelAsync` :

```
bw.WorkerReportsProgress = true;
bw.WorkerSupportsCancellation = true;
```

Le composant contient également trois évènements :

- `DoWork` : cet évènement est déclenché par la méthode `RunWorkerAsync` de l'objet. Il lance l'exécution de la méthode associée à son délégué dans un thread séparé :

```
bw.DoWork += new DoWorkEventHandler(bw_DoWork);
bw.RunWorkerAsync();

void bw_DoWork(object sender, DoWorkEventArgs e)
{
    Library.MailerTools.Send();
}
```

  Le gestionnaire de l'évènement reçoit en paramètre un objet du type `DoWorkEventArgs` avec une propriété `Argument` permettant de transférer des données pour l'exécution de la tâche de fond grâce à une surcharge de la méthode `RunWorkerAsync`.

- `ProgressChanged` : cet évènement est déclenché lors de l'appel de la méthode `ReportProgress` de l'objet. Elle prend en paramètres un entier pour indiquer le pourcentage d'avancement de la tâche et un objet pour transférer tout type de données aux gestionnaires de l'évènement :

```
bw.ProgressChanged +=
          new ProgressChangedEventHandler(bw_ProgressChanged);

void bw_DoWork(object sender, DoWorkEventArgs e)
{
    bw.ReportProgress(0, "Envoi en cours");
```

```
    Library.MailerTools.Send();
}
void bw_ProgressChanged(object sender,
                ProgressChangedEventArgs e)
{
    this.lblStatus.Text = e.UserState.ToString();
}
```

Le gestionnaire de l'évènement reçoit un objet en paramètre du type `ProgressChangedEventArgs` qui expose les valeurs assignées lors de l'appel de la méthode `ReportProgress` c'est-à-dire `ProgressPercentage` et `UserState`.

- `RunWorkerCompleted` : cet évènement est déclenché automatiquement à la fin de l'exécution de la tâche de fond et les gestionnaires de cet évènement peuvent réaliser les opérations souhaitées comme la mise à jour de l'interface utilisateur :

```
bw.RunWorkerCompleted +=
      new RunWorkerCompletedEventHandler(bw_RunWorkerCompleted);

void bw_RunWorkerCompleted(object sender,
                           RunWorkerCompletedEventArgs e)
{
    this.lblStatus.Text = "Envoi terminé";
}
```

Le gestionnaire de l'évènement reçoit un objet en paramètre du type `RunWorkerCompletedEventArgs` qui expose les propriétés `Result` de type `object` permettant de transférer les résultats de la tâche de fond et `UserState` permettant de transférer les données originales.

Le composant `BackgroundWorker` expose également deux propriétés en lecture seule : `CancellationPending` et `IsBusy`.

- `CancellationPending` indique que la méthode `CancelAsync` a été appelée. La tâche de fond doit implémenter une manière d'annuler son exécution élégamment en testant la valeur de cette propriété car le thread ne sera pas supprimé et l'exécution continuera.

- `IsBusy` permet de savoir si la tâche d'arrière-plan est en cours d'exécution ou non. Sa valeur est égale à `true` lorsque la méthode `RunWorkerAsync` est appelée et redevient `false` lorsque le travail est terminé et que l'évènement `RunWorkerCompleted` est déclenché.

Introduction

Bien que très proche dans le but, la globalisation et la localisation d'une application sont deux choses distinctes. La globalisation est le procédé qui permet de mettre en forme les données en fonction de la culture comme la monnaie ou les dates, tandis que la localisation est le procédé qui va afficher des données différentes suivant la culture de l'application comme par exemple le texte des éléments de l'interface utilisateur.

La globalisation et la localisation reposent sur le concept de culture. La culture fait référence aux spécificités d'un pays ou d'une langue. Les classes permettant de gérer la culture se trouvent dans l'espace de noms `System.Globalization`.

La culture

Les différentes cultures disponibles sont identifiées par un code de culture, il est composé de deux caractères indiquant le langage et peut facultativement être suivi de deux autres caractères pour spécifier la région. Ces deux éléments sont séparés par un tiret :

- `fr` est le code de la langue française sans spécification de région.
- `fr-FR` est le code de la langue française en France.
- `fr-CH` est le code de la langue française en Suisse.
- `en` est le code de la langue anglaise sans spécification de région.

Il existe de nombreux codes de culture pour chaque langue et chaque spécificité de région. La liste complète de ces codes est disponible sur le MSDN sous la rubrique **Classe CultureInfo**. Les codes de culture sans spécification de langage sont appelés des cultures neutres tandis que les codes de culture avec indication de la région sont nommés cultures spécifiques.

Par défaut, lorsqu'une application est lancée, les paramètres culturels sont identiques à ceux du système. Il est possible de modifier cette valeur par programmation.

Créez un nouveau formulaire dans le projet **SelfMailer** et nommez-le **ChooseCulture**. Ajoutez un contrôle `ComboBox` et un contrôle `Label` de la manière suivante :

La culture courante est accessible via la propriété statique `CurrentCulture` de la classe `CultureInfo`. Double cliquez sur le formulaire dans le concepteur de vue pour ajouter le gestionnaire de l'évènement `Load` du formulaire et saisissez le code suivant :

```
private void ChooseCulture_Load(object sender, EventArgs e)
{
    this.lblCulture.Text = string.Format("Culture: {0}",
                            CultureInfo.CurrentCulture.Name);
    this.CultureList.Items.Add(CultureInfo.CurrentCulture.Name);
    this.CultureList.Items.Add("fr");
    this.CultureList.Items.Add("fr-FR");
    this.CultureList.Items.Add("fr-CH");
    this.CultureList.Items.Add("en");
    this.CultureList.Items.Add("en-US");
}
```

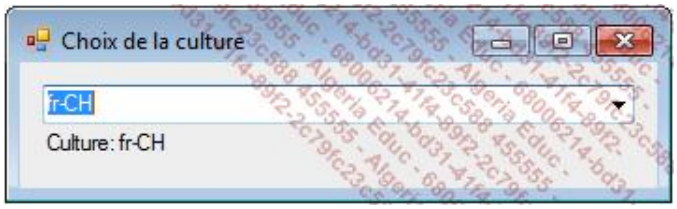
Pour modifier la culture en cours, il suffit d'instancier un objet `CultureInfo` avec le code culture souhaité en paramètre de son constructeur et de l'assigner à la propriété `CurrentCulture` du thread en cours. Ajoutez un gestionnaire à l'évènement `SelectedIndexChanged` du contrôle `ComboBox` :

```
private void CultureList_SelectedIndexChanged(object sender,
EventArgs e)
{
    CultureInfo culture = new CultureInfo(
                    this.CultureList.SelectedItem.ToString());
    Thread.CurrentThread.CurrentCulture = culture;
    this.lblCulture.Text = string.Format("Culture: {0}",
                            CultureInfo.CurrentCulture.Name);
}
```

Modifiez la méthode `Main` dans le fichier **Program.cs** de la manière suivante pour afficher en premier lieu le formulaire de choix de la culture :

```
static void Main()
{
    Project = new Library.Project();
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Forms.ChooseCulture());
    Application.Run(new Forms.Main());
}
```

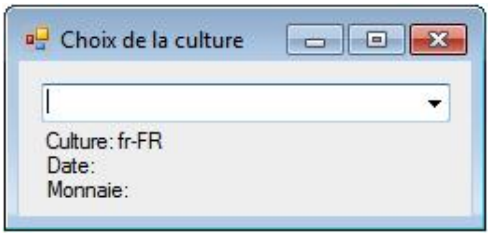
Lancez l'application (**F5**) et sélectionnez différentes valeurs dans la liste pour changer la culture de l'application :



La globalisation

Suivant le pays, la monnaie est mise en forme de différentes manières, certains pays utilisent la virgule comme séparateur décimal alors que d'autres utiliseront le point. Bien évidemment, les données à afficher sont identiques d'un pays à l'autre.

Lorsque la culture est modifiée, toute donnée mise en forme par l'application se verra appliquer automatiquement le nouveau format. Ajoutez deux contrôles `Label` sur le formulaire **ChooseCulture** comme suit :



Pour être correctement affichée en fonction de la culture, une valeur doit utiliser le formatage fourni par le framework .NET. Si par exemple vous utilisez l'instruction suivante pour afficher la date :

```
this.lblDate.Text = DateTime.Now.ToString("dd/MM/yyyy");
```

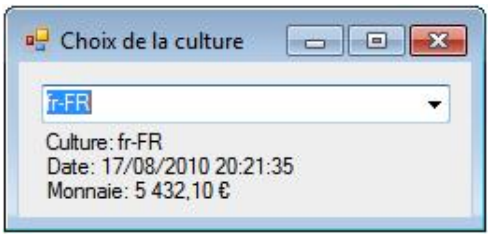
le changement de culture n'aura aucun effet puisque le format est explicitement spécifié. Par contre si vous utilisez l'affichage par défaut, le formatage utilisé sera celui de la culture courante.

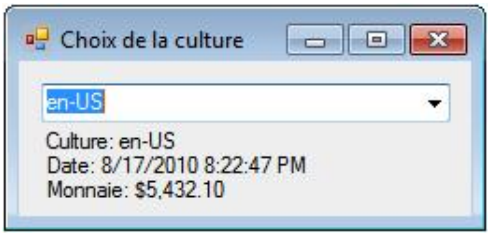
Modifiez le gestionnaire de l'évènement `SelectedIndexChanged` pour modifier la valeur de ces contrôles :

```
this.lblDate.Text = string.Format("Date: {0}",
                                  DateTime.Now.ToString());
this.lblMoney.Text = string.Format("Monnaie: {0}",
                                  (5432.1).ToString("C"));
```

Pour les dates, appeler la méthode `ToString` est suffisant pour mettre en forme la donnée en fonction de la culture courante tandis que pour les nombres, il faut spécifier de quel type de nombre il s'agit. Le paramètre `"C"` indique que le nombre est monétaire et qu'il doit être formaté suivant la culture en cours. La liste complète des paramètres de formatage des nombres est disponible sur le MSDN sous la rubrique **Classe NumberFormatInfo**.

Lancez l'application pour observez le changement de formatage :





Le formatage spécifique à la culture est modifiable à partir du code. Si par exemple vous souhaitez afficher une application avec les paramètres culturels de la France mais que le système monétaire doit être le Dollar Américain, vous pouvez modifier la culture pour modifier le symbole. La classe `CultureInfo` contient des propriétés pour chacune des variables culturelles comme les calendriers (`Calendar`), les formats de date (`DateTimeFormat`) ou encore les formats des nombres (`NumberFormat`). Chacune de ces propriétés comporte des propriétés permettant de définir précisément un point spécifique du formatage. Pour créer une nouvelle culture, il suffit donc d'instancier un nouvel objet `CultureInfo` avec la culture la plus proche de celle que vous souhaitez obtenir de manière à avoir le moins de modifications possible à effectuer. Dans un second temps, modifiez les propriétés adéquates pour correspondre avec la culture recherchée et pour finir, définissez cette culture comme étant celle du thread courant :

```
CultureInfo spec = new CultureInfo("fr-FR");
spec.NumberFormat.CurrencySymbol = "$";
Thread.CurrentThread.CurrentCulture = spec;
```

À noter que le changement de culture et de système monétaire n'entraîne aucune conversion d'une devise à l'autre. Seul l'affichage du nombre est différent.

La localisation

La localisation concerne tous les éléments qui d'une langue à l'autre ou même en fonction du pays vont devoir être modifiés. L'exemple le plus commun est les textes des formulaires. Le Framework .NET met à disposition les fichiers de ressources qui vont stocker les valeurs dans l'assemblage de l'application ou un autre assemblage satellite qui sera référencé dans l'application. C'est à l'exécution que le choix des ressources sera effectué en fonction de la culture.

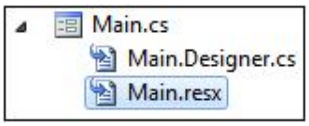
Les fichiers de ressources sont choisis pendant la phase d'exécution en fonction de la propriété `CurrentUICulture` du thread courant. Cette propriété de type `CultureInfo` peut être différente de la propriété `CurrentCulture`. Elles sont toutes les deux distinctes et n'influent pas l'une sur l'autre.

La modification de la culture de localisation par programmation se fait de la même manière que pour la culture de globalisation.

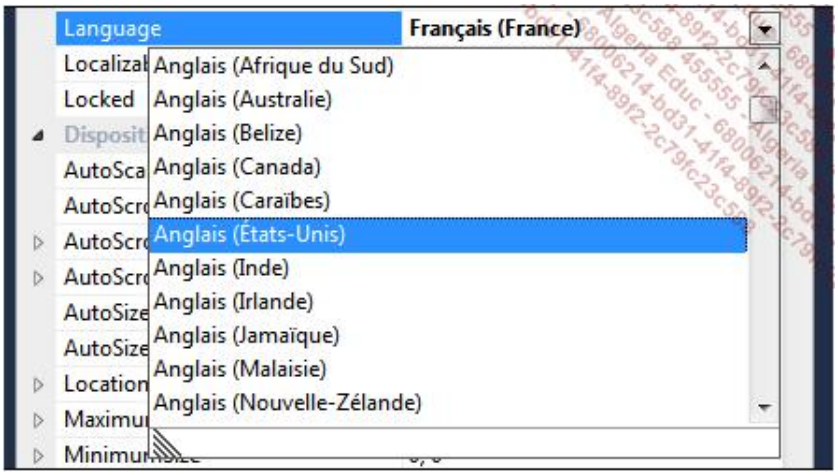
Ajoutez la mise à jour de la propriété `CurrentUICulture` au thread courant dans le gestionnaire de l'évènement du formulaire `ChooseCulture` :

```
Thread.CurrentThread.CurrentUICulture = culture;
```

Pour que cette propriété soit prise en compte, il faut localiser les formulaires. Chacun d'eux possède une propriété `Localizable` avec la valeur par défaut `False`. Ouvrez le formulaire **Main** et modifiez cette valeur par `True` à partir de la fenêtre **Propriétés**. Dans la fenêtre **Explorateur de solutions**, un fichier **Main.resx** a été créé :



En ouvrant ce fichier, vous constatez que toutes les propriétés du formulaire et de ses contrôles qui sont différentes des valeurs par défaut sont référencées. Ce fichier correspond à la culture par défaut. Vous pouvez créer de nouveaux fichiers de ressources en sélectionnant dans la fenêtre **Propriétés** du formulaire la culture souhaitée dans la propriété `Language` :



Sélectionnez la valeur **Anglais (Etats-unis)**. Aucune modification n'est apportée en interne à ce stade car il n'y a aucun contenu localisé qui a été saisi. Modifiez la propriété `Text` du contrôle `fichierToolStripMenuItem` par `&File`. Visual Studio détecte que ce contenu correspond à la culture en-US qui n'est pas la culture par défaut. Il va donc créer un nouveau fichier de ressources pour y stocker les valeurs des propriétés du formulaire :

Les fichiers de ressources sont nommés de la manière suivante :

```
[NomFormulaire].[Culture].resx
```

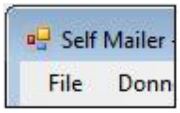
En ouvrant le fichier **Main.en-US.resx**, vous remarquez que seule la propriété précédemment modifiée est référencée. Cela est dû au mode de fonctionnement de la localisation. Lorsqu'un formulaire est chargé, les ressources en fonction

de la culture en cours sont également chargées. Si une propriété n'est pas référencée dans le fichier de ressources, l'application va automatiquement chercher dans le fichier de ressources supérieur, c'est-à-dire le fichier de ressources de la culture neutre puis dans les ressources par défaut. Il n'est donc pas nécessaire d'initialiser toutes les propriétés de tous les contrôles d'un formulaire pour qu'il soit localisé.

Pour localiser un formulaire, il faut donc :

- Définir la propriété `Localizable` du formulaire à `True`.
- Choisir la langue à éditer avec la propriété `Language` du formulaire.
- Modifier toutes les propriétés qui doivent être localisées.
- Répéter les étapes 2 et 3 pour chaque culture.

Lancez l'application (**F5**), choisissez la culture **en-US** dans le formulaire **ChooseCulture** puis fermez-le. Le formulaire **Main** apparaît avec le menu localisé :



Refaites la même procédure en choisissant une autre culture, le menu est affiché avec les paramètres culturels par défaut :

Introduction

Dans une entreprise, la politique de sécurité globale est définie par l'administrateur système et elle ne peut en aucun cas être outrepassée par le code. L'administrateur fixe des rôles à chacun des utilisateurs et groupes à partir d'Active Directory. De cette sécurité basée sur les rôles résultent des autorisations. Ces autorisations peuvent être impératives, c'est-à-dire que la permission est demandée au moment de l'exécution ou cela peut être des autorisations déclaratives, c'est-à-dire que l'assemblage va spécifier les permissions requises pour son exécution. En cas de conflit, l'assemblage et donc l'application ne pourront pas s'exécuter.

Les éléments de base

1. L'interface IPermission

Les classes définissant les permissions sont exposées dans l'espace de noms `System.Security.Permissions` et sont au centre du processus de sécurité. Il existe différentes classes pour différents types de permissions. Par exemple, l'accès au système de fichiers sera géré par la classe `FileIOPermission`, l'accès aux variables du registre par la classe `RegistryPermission` et l'accès aux variables d'environnement par la classe `EnvironmentPermission`.

Toutes ces classes implémentent l'interface `IPermission` qui implémente les méthodes communes aux permissions :

- `Copy` : retourne une copie de l'autorisation en cours.
- `Demand` : effectue une vérification des permissions et lève une exception du type `SecurityException` si les conditions ne sont pas réunies.
- `Intersect` : retourne une nouvelle autorisation qui représente l'intersection de l'autorisation en cours et de celle spécifiée en paramètre.
- `IsSubsetOf` : retourne une valeur booléenne indiquant si l'autorisation en cours est un sous-ensemble de celle spécifiée en paramètre.
- `Union` : retourne une nouvelle autorisation qui représente l'union de l'autorisation en cours et de celle spécifiée en paramètre.

2. La classe CodeAccessPermission

Toutes les permissions d'accès héritent de la classe abstraite `CodeAccessPermission` qui implémente l'interface `IPermission`. Par conséquent, elles exposent toutes un jeu de membres commun permettant à la fois de vérifier et d'appliquer une politique de sécurité :

- `Assert` : déclare que le code appelant peut accéder à la ressource même si les appelants plus hauts dans la pile n'ont pas de permission.
- `PermitOnly` : déclare que le code appelant n'a pas la permission d'accéder à la ressource à l'exception du sous-ensemble spécifié par la permission.
- `RevertAll` : entraîne la suppression de toutes les politiques définies précédemment avec les méthodes `Assert`, `PermitOnly`.
- `RevertAssert` : entraîne la suppression de toutes les politiques définies précédemment avec la méthode `Assert`.
- `RevertPermitOnly` : entraîne la suppression de toutes les politiques définies précédemment avec la méthode `PermitOnly`.

3. L'interface IPrincipal

Les classes définissant les utilisateurs sont exposées dans l'espace de noms `System.Security.Principal`. Elles implémentent l'interface `IPrincipal` qui sert de base pour la définition d'un utilisateur avec une méthode `IsInRole` qui retourne une valeur booléenne déterminant si l'utilisateur appartient au rôle spécifié et une propriété `Identity` qui implémente l'interface `IIdentity`.

L'interface `IIdentity` expose en lecture seule les propriétés de base que doit implémenter un objet d'identité :

- `AuthenticationType` : retourne le type d'authentification utilisé. Par exemple NTLM.

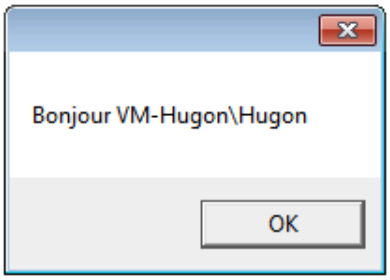
- `IsAuthenticated` : retourne une valeur booléenne indiquant si l'utilisateur est authentifié.
- `Name` : retourne le nom de l'utilisateur en cours.

Lors du développement d'une application Windows, vous pouvez utiliser le système de sécurité intégré de Windows dans votre application. L'avantage est d'autant plus important s'il s'agit d'une application d'entreprise qui possède un Active Directory et donc un système central de gestion des autorisations. Pour l'utiliser, il suffit de définir la politique principale pour le domaine applicatif en cours avec la valeur `WindowsPrincipal` de l'énumération `PrincipalPolicy` :

```
AppDomain.CurrentDomain.SetPrincipalPolicy(
                            PrincipalPolicy.WindowsPrincipal);
```

Vous pouvez ensuite récupérer une instance d'objet `WindowsPrincipal` définissant l'utilisateur courant :

```
WindowsPrincipal currentUser = Thread.CurrentPrincipal
                             as WindowsPrincipal;
MessageBox.Show("Bonjour " + currentUser.Identity.Name);
```



Le nom d'utilisateur Windows est toujours précédé du nom de la machine ou du domaine Active Directory. Dans ces exemples, les utilisateurs sont des comptes locaux de la machine VM-Hugon.

Implémentation de la sécurité

1. La sécurité basée sur les rôles

La sécurité basée sur les rôles permet d'accorder ou de refuser l'accès à une partie de l'application ou des ressources en fonction de l'identité de l'utilisateur. Le principe de base est d'avoir, d'un côté un utilisateur et de l'autre une permission. L'accès se fait seulement si l'utilisateur satisfait aux conditions de la permission.

a. Sécurité impérative

La sécurité impérative permet de facilement gérer l'accès aux différentes parties d'une application. Prenons l'exemple de l'application **SelfMailer**. Vous pouvez par exemple définir que seul un administrateur peut utiliser le menu **Envoyer**. Pour réaliser cette tâche, vous devez commencer par créer un nouvel objet `PrincipalPermission` en spécifiant une identité et/ou un rôle dans le constructeur du formulaire **Main** :

```
PrincipalPermission permission = new PrincipalPermission(
                                    @"VM-Hugon\Hugon",
                                    "Administrators");
```

Vous pouvez ensuite lancer la vérification de la permission en appelant la méthode `Demand`. Cette méthode génère une exception du type `SecurityException` si l'utilisateur n'a pas les autorisations nécessaires à la permission :

```
try
{
    permission.Demand();
}
catch (SecurityException ex)
{
    this.envoyerToolStripMenuItem.Visible = false;
}
```

Dans le cas où plusieurs utilisateurs devraient avoir l'autorisation d'accéder à une partie de l'application, il est possible d'utiliser la méthode `Union` pour créer une nouvelle permission en combinant deux autres :

```
PrincipalPermission permission1 = new PrincipalPermission(
                                    @"VM-Hugon\Hugon",
                                    "Administrators");
PrincipalPermission permission2 = new PrincipalPermission(
                                    @"VM-Hugon\Administrator",
                                    "Administrators");
PrincipalPermission permission = permission1.Union(permission2)
                                 as PrincipalPermission;
```

Le fait de spécifier une valeur `null` pour le nom ou le rôle permet de créer une permission ne validant que le nom ou le rôle de l'utilisateur :

```
PrincipalPermission permission = new PrincipalPermission(
                                    null,
                                    "Administrators");
```

L'exemple précédent crée une permission autorisant tous les utilisateurs du groupe `Administrators`.

La méthode `Intersect` va créer une nouvelle permission représentant l'intersection de deux autres :

```
PrincipalPermission permission1 = new PrincipalPermission(
                                    null,
                                    "Administrators");
PrincipalPermission permission2 = new PrincipalPermission(
                                    @"VM-Hugon\GuestUser",
                                    null);
PrincipalPermission perm = permission1.Intersect(permission2)
                           as PrincipalPermission;
```

Pour satisfaire la permission de l'exemple précédent, l'utilisateur devra se nommer `GuestUser` et faire partie du

groupe Administrators.

b. Sécurité déclarative

Tous les objets permissions possèdent un attribut correspondant. Ils sont appliqués à une classe ou une méthode de manière à contrôler l'accès aux portions de code. Dans la sécurité déclarative, les attributs protègent les classes et méthodes et sont transmis dans les métadonnées du type. L'administrateur système peut ainsi consulter les métadonnées de l'assemblage pour décider ou non d'autoriser une application à s'exécuter.

Chacun des attributs de permission prend comme paramètre de son constructeur une valeur de l'énumération SecurityAction permettant de spécifier quelle action effectuer pour vérifier l'autorisation. Les propriétés concernant le rôle ou l'utilisateur sont également définies dans le constructeur. Pour sécuriser la classe SelfMailer.Forms.Send afin qu'elle ne puisse être exécutée que par le groupe Administrators, vous pouvez appliquer l'attribut PrincipalPermissionAttribute sur la classe de la manière suivante :

```
[PrincipalPermission(SecurityAction.Demand,
                     Role = "Administrators")]
public partial class Send : Form
{ ... }
```

Si l'utilisateur qui essaie d'exécuter ce code ne satisfait pas aux conditions de la permission, une exception du type SecurityException sera levée.

2. La sécurité basée sur les droits d'accès

La sécurité basée sur les droits d'accès repose sur les permissions mais contrairement à la sécurité basée sur les rôles où une permission représente un utilisateur, une permission va représenter une ressource système. Les ressources peuvent être le système de fichiers, le registre, une imprimante ou encore une base de données.

a. Sécurité impérative

Dans le cadre de la sécurité impérative, les permissions sont définies et appliquées au moment de l'exécution. La permission d'accéder à une ressource est donnée par le CLR. Il vérifie la politique d'accès définie par l'administrateur système pour l'assemblage et parcourt la pile des appels pour vérifier que chacun des appelants a obtenu la permission d'accéder à la ressource.

Si l'application doit pouvoir écrire sur le système de fichiers, pour s'assurer que la politique de sécurité autorise l'accès à cette ressource, vous devez instancier un objet FileIOPermission et demander sa vérification :

```
FileIOPermission permission = new FileIOPermission(
                                  PermissionState.Unrestricted);
permission.Demand();
```

L'énumération PermissionState comporte deux valeurs : Unrestricted pour créer une permission avec un accès non restreint à la ressource et None pour créer une permission avec absence de droits d'accès à la ressource.

Lorsque la méthode Demand est appelée, elle parcourt la pile des appels pour vérifier que tous les appelants ont obtenu la permission d'accéder à la ressource.

Les classes de permissions vous permettent de préciser assez finement le degré de permission. Pour la classe FileIOPermission, vous pouvez par exemple préciser quel type d'accès doit être autorisé et sur quel fichier :

```
FileIOPermission permission = new FileIOPermission(
                                  FileIOPermissionAccess.Write,
                                  @"C:\file.txt");
```

La méthode PermitOnly permet d'interdire l'accès à une ressource sauf pour la permission définie. L'exemple précédent définit un droit d'écriture sur un fichier. En exécutant la méthode PermitOnly de cette permission, toute tentative d'écriture échouera sauf pour le fichier spécifié :

```
permission.PermitOnly();
```

La méthode Assert déclare qu'une méthode a la permission d'accéder à une ressource. Cette méthode peut être dangereuse car contrairement à la méthode Demand, la pile des appels n'est pas parcourue. La demande d'accès peut ainsi approuver du code qui ne le serait pas par un appelant. La méthode Assert ne pourra en aucun cas outrepasser la politique de sécurité du système :

```
permission.Assert();
```

Les méthodes statiques `RevertAll`, `RevertAssert` et `RevertPermitOnly` annulent toutes les permissions précédemment attribuées sur la ressource :

```
FileIOPermission.RevertAll();
FileIOPermission.RevertAssert();
FileIOPermission.RevertPermitOnly();
```

b. Sécurité déclarative

Comme pour la sécurité basée sur les rôles, vous pouvez faire appel à la sécurité déclarative pour les droits d'accès. Chaque permission d'accès possède un attribut correspondant qui peut être utilisé pour marquer une classe ou une méthode afin de définir les actions de sécurité.

Pour associer un attribut de permission, la méthodologie est identique à celle de la sécurité basée sur les rôles en spécifiant une valeur de l'énumération `SecurityAction` qui correspond à l'une des méthodes de la permission concernée ou à une des deux actions de sécurité supplémentaires `LinkDemand` et `InheritanceDemand`. `LinkDemand` exige que l'appelant immédiat ait reçu la permission spécifiée tandis que `InheritanceDemand` exige que la classe dérivée ou la méthode qui substitue l'originale doit avoir reçu les autorisations spécifiées. L'exemple suivant illustre comment définir une classe pour exiger que toutes les classes qui en dériveront devront bénéficier de la permission `FileIOPermission` :

```
[FileIOPermission(SecurityAction.InheritanceDemand)]
public class maClasse
{ ... }
```

Introduction à la cryptographie

Les données sensibles doivent être sécurisées de manière à ce que seuls les utilisateurs autorisés puissent les consulter. L'encryptage est une manière de sécuriser les informations. Il existe des algorithmes de cryptage symétriques et d'autres, asymétriques. Un algorithme de cryptage symétrique utilise la même clé pour l'encryptage et pour le décryptage tandis qu'un algorithme de cryptage asymétrique utilisera des clés différentes pour l'encryptage et pour le décryptage : une clé publique et une clé privée. Des données peuvent ainsi être encryptées avec la clé privée et décryptées avec la clé publique ou inversement mais jamais avec la même clé. Le Framework .NET contient de nombreuses classes permettant de faire de la cryptographie. Elles sont exposées dans l'espace de noms `System.Security.Cryptography`. Le hachage est une autre technique de cryptographie mais son but n'est pas de sécuriser les données mais de garantir leur intégrité. Les algorithmes de hachage ont pour but de créer une valeur de longueur fixe à partir d'une source de longueur variable. Ces algorithmes sont utilisés pour les signatures numériques et pour vérifier l'intégrité des données. Une donnée passée dans un algorithme de hachage donnera toujours le même résultat.

L'enregistrement des projets de l'application **SelfMailer** se fait au format XML. Le mot de passe du compte email est donc humainement lisible. Nous allons voir comment mettre en œuvre l'encryptage et le décryptage de cette propriété.

Créez une nouvelle classe statique `Cryptor` dans le dossier **Library** du projet :

```
public static class Cryptor
{
}
```

La classe `DESCryptoServiceProvider` sera utilisée. Cette méthode d'encryptage est considérée comme non sécurisée car la clé peut être cassée en moins de 24 heures mais ce sera suffisant pour exposer le principe de la cryptographie. Pour réaliser les opérations d'encryptage et de décryptage, avec un objet `DESCryptoServiceProvider`, vous devez fournir deux tableaux de type `byte` d'une longueur de 8 éléments chacun. L'un d'eux sera affecté à la propriété `Key` et l'autre à la propriété `IV` correspondant au vecteur. Ajoutez deux membres à la classe `Cryptor` définissant un tableau de 8 bits nommé `Key` et un second nommé `Vector` :

```
private static byte[] Key = new byte[]
{ 140, 58, 74, 45, 196, 24, 19, 220 };

private static byte[] Vector = new byte[]
{ 211, 26, 16, 198, 172, 15, 1, 3 };
```

Les valeurs de la clé et du vecteur doivent être identiques lors de l'opération de décryptage à celles qui ont été utilisées pour l'opération d'encryptage.

Vous pouvez maintenant créer un objet `DESCryptoServiceProvider`, initialiser sa clé et son vecteur afin d'obtenir une instance d'objet `ICryptoTransform`. C'est cet objet qui va réaliser le cryptage des données grâce à sa méthode `TransformFinalBlock` :

```
public static byte[] SwitchCrypt(byte[] data)
{
    DESCryptoServiceProvider o = new DESCryptoServiceProvider();
    o.Key = Key;
    o.IV = Vector;
    ICryptoTransform cryptor = o.CreateEncryptor();
    return cryptor.TransformFinalBlock(data, 0, data.Length);
}
```

Le cryptage avec l'objet `DESCryptoServiceProvider` étant symétrique, la méthode pour l'encryptage et le décryptage est identique.

Créez une surcharge de la méthode `SwitchCrypt` prenant en charge les objets de type `string` :

```
public static string SwitchCrypt(string s)
{
    byte[] original = Encoding.UTF8.GetBytes(s);
    byte[] switched = Cryptor.SwitchCrypt(original);
    return Encoding.UTF8.GetString(switched);
}
```

Pour finir la sécurisation du mot de passe, ajoutez l'attribut `XmlIgnore` sur la propriété `Password` de la classe `MailServerSettings` et ajoutez une nouvelle propriété publique :

```
public string CryptedPassword
```

```
        {
            get { return Cryptor.SwitchCrypt(this.password); }
            set { this.password = Cryptor.SwitchCrypt(value); }
        }
```

La propriété `Password` ne sera plus sérialisée et la propriété `CryptedPassword` qui sera sérialisée agit comme une couche intermédiaire permettant d'encrypter et décrypter la valeur de la variable `password` :

```
<?xml version="1.0"?>
<Project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ProjectSettings>
    ...
  </ProjectSettings>
  <MailServerSettings>
    <FromName />
    <FromEmail />
    <Host />
    <Username />
    <CryptedPassword>  Ä*   M'|  @</CryptedPassword>
  </MailServerSettings>
  <MailProperties>
    ...
  </MailProperties>
  <Data>
    ...
  </Data>
</Project>
```

Le dessin avec GDI+

Lors du développement d'application, la couche d'abstraction de Windows permet de ne pas avoir à écrire du code pour un composant particulier. Par exemple, pour les cartes graphiques, il en existe de très nombreuses, toutes différentes. C'est pourquoi la couche GDI (*Graphics Device Interface*) est mise à disposition des développeurs. Ils communiquent avec cette API et elle fera ensuite le lien avec Windows puis les informations passeront par le driver de la carte graphique et enfin l'écran. Dans le Framework .NET, cette API est connue sous le nom de GDI+, elle est répartie sur six espaces de noms :

- `System.Drawing` : cet espace de noms contient la plupart des classes, structures et autres définitions de base pour la programmation graphique.
- `System.Drawing.Design` : cet espace de noms contient les classes utiles pour le développement de l'interface graphique au moment de la conception dans Visual Studio.
- `System.Drawing.Drawing2D` : cet espace de noms fournit la plupart des classes pour le dessin en deux dimensions avec notamment le dessin vectoriel, les transformations géométriques et les effets visuels tel que l'anti-aliasing.
- `System.Drawing.Imaging` : cet espace de noms fournit les classes permettant de gérer la manipulation des images.
- `System.Drawing.Printing` : cet espace de noms fournit des classes permettant de simplifier l'impression de contenu texte et image.
- `System.Drawing.Text` : cet espace de noms contient les classes qui simplifient la manipulation des polices de caractères.

1. La classe Graphics

La classe `Graphics` est celle principalement utilisée dans l'affichage. Un objet `Graphics` représente une surface de dessin pour un élément. Cela peut être un formulaire, un bouton ou un label. Cette classe réalise l'opération d'affichage des éléments visuels.

Un objet `Graphics` ne peut pas être instancié directement avec son constructeur. Il doit être créé directement à partir de l'élément visuel. Les classes dérivant de la classe `Control` exposent la méthode `CreateGraphics` qui renvoie une référence à l'objet `Graphics` associé :

```
Forms.Main main = new Forms.Main();
System.Drawing.Graphics graphics = main.CreateGraphics();
```

Pour modifier le rendu du contrôle ou du formulaire, il suffit alors de modifier l'objet `Graphics` ainsi obtenu.

Si vous travaillez sur des images, la classe `Graphics` expose la méthode statique `FromImage` pour récupérer l'instance de l'objet `Graphics` associée à une image :

```
Bitmap bitmap = new Bitmap(@"C:\image.bmp");
Graphics graphics = Graphics.FromImage(bitmap);
```

a. Les coordonnées

L'affichage des éléments graphiques se fait toujours sur une surface rectangulaire dont l'origine se situe par défaut dans le coin supérieur gauche dont les coordonnées en pixels sont (0, 0). Les structures suivantes permettent de décrire des emplacements ou des régions de formes dans le système de coordonnées :

- `Point` : cette structure représente un point unique avec des coordonnées `X` et `Y` de type `int`.

```
System.Drawing.Point point = new System.Drawing.Point(10, 20);
```

- `Size` : cette structure représente la taille avec une valeur `Height` et une valeur `Width` de type `int`.

```
System.Drawing.Size size = new System.Drawing.Size(50, 50);
```

- `Rectangle` : cette structure représente une région rectangulaire avec une position représentée par une structure `Point` et une taille représentée par une structure `Size`.

```
System.Drawing.Rectangle rectangle = new Rectangle(point, size);
```

Chacune de ces trois structures possède son équivalent reposant sur des types `float`: `PointF`, `SizeF` et `RectangleF`. La conversion d'une structure de type `int` vers une structure équivalente de type `float` se fait implicitement. La conversion inverse doit être effectuée explicitement.

b. Les formes

L'objet `Graphics` expose de nombreuses méthodes permettant de réaliser des formes simples et complexes sur la surface de dessin. Celles préfixées par `Draw`, sont destinées au dessin de structures linéaires comme des lignes, des arcs ou des contours de formes rectangulaires ou circulaires. Les méthodes préfixées par `Fill` sont destinées à être utilisées pour dessiner des formes pleines.

Le dessin d'une forme simple a déjà été utilisé lors de la création du contrôle personnalisé `CustomControl` :

```
protected override void
OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    Rectangle R = new Rectangle(0,
                                0,
                                this.Size.Width,
                                this.Size.Height);
    e.Graphics.FillRectangle(Brushes.Green, R);
}
```

Voici quelques-unes des méthodes exposées par la classe `Graphics` pour la création de formes :

- `DrawArc` : dessine un arc représentant la portion d'une ellipse dans un rectangle.
- `DrawBezier` : dessine une courbe de Béziers.
- `DrawBeziers` : dessine une série de courbes de Béziers.
- `DrawClosedCurve`/ `FillClosedCurve` : dessine une courbe fermée basée sur une série de points.
- `DrawCurve` : dessine une courbe ouverte basée sur une série de points.
- `DrawEllipse`/ `FillEllipse` : dessine une ellipse s'inscrivant dans le rectangle dans lequel elle est définie.
- `DrawLine` : dessine une ligne entre deux points.
- `DrawLines` : dessine une série de lignes reliant les points d'un tableau.
- `DrawPath`/ `FillPath` : dessine un objet `GraphicsPath` représentant une forme complexe.
- `DrawPie`/ `FillPie` : dessine un secteur défini par une ellipse.
- `DrawPolygon`/ `FillPolygon` : dessine un polygone à partir d'une série de points.
- `DrawRectangle`/ `FillRectangle` : dessine un rectangle.
- `DrawRectangles`/ `FillRectangles` : dessine une série de rectangles.

- `DrawString` : dessine une chaîne de caractères dans un rectangle défini dans la police spécifiée.

Chacune des méthodes prend comme arguments un ou plusieurs jeux de coordonnées spécifiant l'emplacement des formes ainsi qu'un objet `Pen` pour les formes linéaires ou un objet `Brush` pour les formes pleines.

2. La structure Color et les classes Brush et Pen

Les classes `Brush` et `Pen` déterminent de quelle manière les éléments graphiques seront dessinés. La grande différence entre ces deux classes est que la classe `Brush` sert à afficher des formes pleines tandis que la classe `Pen` est utilisée pour l'affichage de formes linéaires. La structure `Color`, permet de spécifier qu'elle sera la couleur de la forme.

a. La structure Color

La structure `Color` est composée de quatre valeurs de base : `A` pour la composante alpha, c'est-à-dire la transparence de la couleur, `R` pour la composante rouge, `G` pour la composante verte et `B` pour la composante bleue. Chacune de ces propriétés est de type `byte` permettant de définir une valeur comprise entre 0 et 255. La méthode `FromArgb` permet de créer une couleur personnalisée. L'exemple suivant montre la création d'une couleur verte pure avec 50% de transparence :

```
Color color = Color.FromArgb(128, 0, 255, 0);
```

Une surcharge de la méthode `FromArgb` permet de ne pas avoir à spécifier la composante alpha de la couleur dans le cas où aucune transparence n'est requise :

```
Color color = Color.FromArgb(0, 255, 0);
```

La structure `Color` contient également de nombreuses propriétés en lecture seule retournant des couleurs nommées :

```
Color color;
color = Color.Black;
color = Color.Brown;
color = Color.Orchid;
color = Color.Salmon;
color = Color.White;
```

b. La classe Brush

La classe abstraite `Brush` permet de définir le remplissage des formes pleines. Les classes qui en dérivent permettent de modifier le style de l'affichage. La liste suivante décrit les classes dérivées de `Brush` plus courantes :

- `SolidBrush` : dessine la forme en la remplissant avec une couleur unie.
- `TextureBrush` : dessine la forme en la remplissant avec une image.
- `HatchBrush` : dessine la forme en la remplissant avec un motif de hachures.
- `LinearGradientBrush` : dessine la forme en la remplissant avec un dégradé linéaire de deux couleurs.
- `PathGradientBrush` : dessine la forme en la remplissant avec un dégradé complexe de plusieurs couleurs.

Voici un exemple de création d'un objet `SolidBrush` à partir d'une couleur connue :

```
SolidBrush solidBrush = new SolidBrush(Color.Black);
```

Suivant le type d'objet `Brush` à créer, le constructeur prend différents paramètres.

c. La classe Pen

La classe Pen permet de définir la couleur des formes linéaires, cette classe ne peut pas être dérivée. La création d'un nouvel objet Pen prend en paramètre une couleur :

```
Pen pen = new Pen(Color.Black);
```

Vous pouvez modifier la largeur du trait en spécifiant une nouvelle largeur dans le constructeur ou en assignant la nouvelle valeur à la propriété Width :

```
Pen pen = new Pen(Color.Black, 2);
pen.Width = 5;
```

Le constructeur accepte également un objet Brush à la place d'une structure Color. L'objet Pen créé possèdera ainsi la même couleur que celle de l'objet Brush :

```
SolidBrush solidBrush = new SolidBrush(Color.Black);
Pen pen = new Pen(solidBrush);
```

La classe Pen expose de nombreuses propriétés permettant d'affiner l'affichage des lignes comme le style de pointillés avec la propriété DashStyle exposant une valeur de l'énumération DashStyle (Custom, Dash, DashDot, DashDotDot, Dot, Solid) ou la propriété DashPattern pour la définition de pointillés personnalisés.

d. Les paramètres systèmes

Lors de la conception de l'interface utilisateur, vous pouvez souhaiter que votre application soit conforme à la palette de couleurs définie par l'utilisateur de la machine sur laquelle elle s'exécute. Pour faciliter la tâche, le Framework .NET met à disposition, dans l'espace de noms System.Drawing, différentes classes permettant de définir les couleurs des éléments graphiques en fonction de celles du système : SystemColors, SystemBrushes, SystemPens, SystemFonts et SystemIcons.

L'exemple suivant montre comment utiliser la couleur par défaut des boutons défini par Windows :

```
Color color = SystemColors.ButtonFace;
```

3. Les exemples

a. L'affichage de texte

L'exemple suivant utilise le contrôle personnalisé CustomControl créé précédemment dans le dossier **Controls** du projet **SelfMailer**.

L'affichage de texte est réalisé avec la méthode DrawString d'un objet Graphics. L'exemple suivant montre l'affichage d'une chaîne de caractères dans l'espace utile du contrôle de manière à ce que le texte soit complètement affiché et qu'il ne déborde pas :

```
protected override void
OnPaint(System.Windows.Forms.PaintEventArgs e)
{
  string s = "Bonjour !";
  Font font = new Font("Arial", 50, FontStyle.Italic);
  while ((e.Graphics.MeasureString(s, font).Height >
e.ClipRectangle.Height
          || e.Graphics.MeasureString(s, font).Width >
e.ClipRectangle.Width
          ) && font.Size >= 1)
  {
    font = new Font("Arial", font.Size - 1, FontStyle.Italic);
  }
  e.Graphics.DrawString(s, font, Brushes.Black, 0, 0);
}
```

La partie importante du code est l'utilisation de la méthode MeasureString de l'objet Graphics pour déterminer la longueur et la hauteur du rectangle dans lequel s'inscrira la chaîne de caractères en fonction de la police sélectionnée. Une boucle est effectuée jusqu'a ce que la taille du texte soit correcte pour qu'il soit visible

entièrement dans l'espace du contrôle déterminé par la propriété `ClipRectangle`.

Pour tester le contrôle, déposez-le sur un nouveau formulaire **GDISamples** :

b. Redimensionner une image

L'API GDI+ permet de travailler avec les images. L'exemple suivant présente une méthode statique permettant de redimensionner une image :

```
public static void Resize(string ImagePath, Size NewSize)
{
  FileInfo original = new FileInfo(ImagePath);
  if (original.Exists)
  {
    using (Bitmap bitmap = new Bitmap(NewSize.Width,
                                      NewSize.Height))
    {
      using (Graphics graphics = Graphics.FromImage(bitmap))
      {
        using (Image image = Image.FromFile(original.FullName))
        {
          Rectangle rectangle = new Rectangle(0,
                                              0,
                                              NewSize.Width,
                                              NewSize.Height);
          graphics.DrawImage(image, rectangle);
        }
        bitmap.Save(ImagePath);
      }
    }
  }
}
```

Cette méthode prend deux arguments : le premier correspond au chemin de l'image et le second à la nouvelle taille que l'image devra avoir.

Un objet `Bitmap` est d'abord créé avec la nouvelle taille de l'image. La seconde étape consiste à créer un objet `Graphics` à partir de l'objet `Bitmap` précédemment créé. Un dernier objet `Image` est créé avec la méthode statique `FromFile` permettant d'instancier un objet `Image` à partir d'un fichier. L'image est ensuite dessinée sur l'objet `Graphics` avec la méthode `DrawImage` et un gabarit sous la forme d'un objet `Rectangle` aux dimensions voulues. Pour finir, l'image est sauvée sur disque en appelant la méthode `Save` de l'objet `Bitmap`.

À noter l'utilisation des clauses `using` pour automatiquement libérer les ressources des objets `Bitmap`, `Graphics` et `Image`. Étant donné que l'image est sauvegardée à son emplacement d'origine, l'objet `Image` doit être libéré pour que le fichier physique soit libéré en écriture. C'est la raison pour laquelle la méthode `Save` de l'objet `bitmap` est appelée après la fermeture de la clause `using` de l'objet `Image`, afin que les ressources soient libérées.

Le remoting

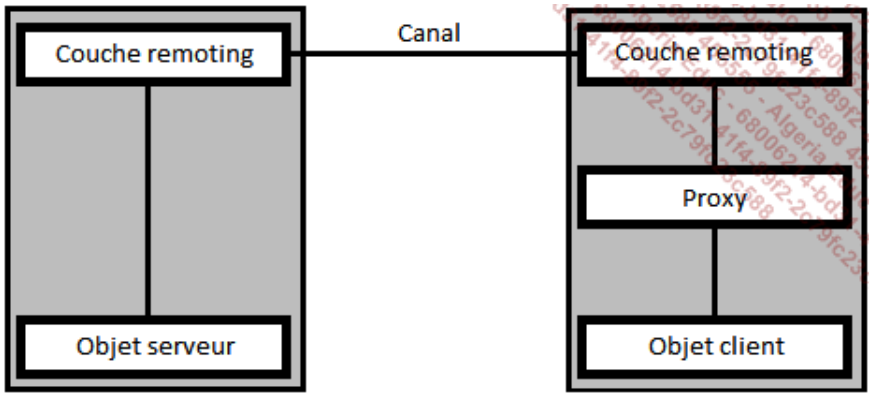
Le remoting permet d'utiliser l'accès distant dans vos applications afin qu'elles communiquent entre elles. Cela peut-être des applications situées sur la même machine, ou sur des machines différentes qui appartiennent soit au même réseau soit à des réseaux différents.

1. Le principe

Le principe du remoting est de faire transiter au travers du réseau des objets, ce principe repose sur l'empaquetage des objets (marshalling et unmarshalling). Le marshalling est le principe de transformation d'un objet pour qu'il puisse transiter au travers d'applications. Le unmarshalling est le processus inverse. Le Framework .NET fournit trois protocoles de formatage dans l'espace de noms `System.Runtime.Remoting.Channels` :

- TCP (`System.Runtime.Remoting.Channels.Tcp`) avec un empaquetage binaire.
- HTTP (`System.Runtime.Remoting.Channels.Http`) avec un empaquetage SOAP.
- IPC (`System.Runtime.Remoting.Channels.Ipc`), ce protocole de communications est plus rapide que les protocoles TCP et HTTP mais il ne peut être utilisé que pour des communications entre applications sur une même machine.

Le schéma suivant illustre la manière dont les applications distantes communiquent :



Le proxy est une couche intermédiaire créée et gérée par le Framework .NET. Il se charge de router les demandes du client vers le serveur et inversement pour récupérer les réponses. Les communications ne se font donc pas directement avec le serveur ou le client mais avec le proxy.

2. L'implémentation

Nous allons construire une application de chat simpliste dont le fonctionnement client/serveur se prête tout à fait aux principes du remoting. La première chose lors de la création d'une application client/serveur est de fournir une interface permettant d'exposer les membres à la fois au client et au serveur. Ensuite il y a la création du serveur et du client en deux applications distinctes.

a. La couche commune

Pour que le client puisse communiquer avec le serveur, il doit connaître les méthodes disponibles. Les classes communes devront donc être distribuées sur l'application cliente et l'application serveur.

Pour simplifier la distribution de la couche commune, nous allons créer une librairie.

Créez un nouveau projet **Bibliothèque de classes** nommé **RemotingLibrary** dans la solution, déclarez la classe `Chat` de la manière suivante :

```
using System;

namespace RemotingLibrary
```

```
{
    public class Chat : MarshalByRefObject
    {
        public override object InitializeLifetimeService()
        {
            return null;
        }
        public bool TransfertMessage(string message)
        {
            Console.WriteLine(String.Format("Message reçu: {0}",
message));
            return true;
        }
    }
}
```

La classe `Chat` est très simple. Elle contient un membre qui sera utilisé pour le transfert des messages, `TransfertMessage`. Cette méthode affiche le message reçu dans la console et retourne la valeur `true`.

La classe `Chat` dérive de la classe abstraite `MarshalByRefObject` qui permet de définir que le client ne contiendra qu'une référence de l'objet. Cela peut être assimilé à un pointeur distant.

La classe `Chat` implémente aussi une surcharge de la méthode `InitializeLifetimeService` en retournant `null`. Cela permet de spécifier que la durée de vie de l'objet est infinie. Définir la durée de vie d'un objet serveur est surtout intéressant s'il y a utilisation de ressources qui doivent être libérées.

b. L'application serveur

Créez un nouveau projet **Application console** nommé **RemotingServer** dans la solution et ajoutez une référence au projet **RemotingLibrary**. Pour accéder aux classes de création de canaux, vous devez ajouter une référence à l'assemblage **System.Runtime.Remoting**.

Modifiez la classe `Program` de la manière suivante :

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemotingServer
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        TcpChannel channel = new TcpChannel(10001);
        ChannelServices.RegisterChannel(channel, false);
        RemotingConfiguration.RegisterWellKnownServiceType(
typeof(Chat), "Chat",  WellKnownObjectMode.Singleton);

        Console.WriteLine("Le serveur a démarré avec succès et
écoute le port 10001.");
        Console.ReadLine();
      }
      catch
      {
        Console.WriteLine("Erreur lors du démarrage du serveur.");
        Console.ReadLine();
      }
    }
  }
}
```

Cette portion de code crée un nouveau canal de type `TcpChannel` sur le port 10001 puis il est enregistré sur le serveur avec la méthode statique `RegisterChannel` de la classe `ChannelServices`. Pour finir, l'écoute est lancée avec l'objet `Chat` en mode `Singleton`, c'est-à-dire qu'il existe une seule instance de l'objet qui sera partagée entre tous les clients. Un nouvel objet sera instancié seulement lorsque la durée de vie du précédent sera expirée. L'autre

mode d'exposition défini par l'énumération `WellKnownObjectMode` est `SingleCall`. Cela signifie que chaque appel entraîne l'instanciation d'un nouvel objet qui est détruit après utilisation. Il est également possible de passer par une activation côté client de l'objet serveur et ainsi d'obtenir une instance d'objet par client.

Pour configurer le serveur, vous pouvez également utiliser la méthode statique `Configure` de la classe `RemotingConfiguration` qui prend en paramètre le nom du fichier de configuration :

```
RemotingConfiguration.Configure("server.config", false);
```

Le fichier de configuration **server.config** est au format XML et comprend toutes les informations permettant de créer les canaux et d'instancier les objets serveurs :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="Chat, RemotingLibrary"
                   objectUri="Chat" />
      </service>
      <channels>
        <channel ref="tcp"
                 port="10001" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

c. L'application cliente

L'application cliente doit commencer par se connecter au serveur distant avec le même protocole et le même port défini dans l'application serveur, puis elle obtiendra une référence de l'objet distant pour exécuter les méthodes sur celui-ci.

Créez un nouveau projet **Application console** nommé **RemotingClient** dans la solution et ajoutez une référence au projet **RemotingLibrary**. Pour accéder aux classes de création de canaux, vous devez ajouter une référence à l'assemblage `System.Runtime.Remoting`.

La connexion au serveur se fait en instanciant un nouveau canal puis en se connectant à l'adresse du serveur :

```
TcpChannel channel = new TcpChannel();
ChannelServices.RegisterChannel(channel, false);
Chat chat = (Chat)Activator.GetObject(typeof(Chat),
                            "tcp://localhost:10001/Chat");
```

On récupère ainsi une instance d'un objet distant mais qui va pouvoir être manipulé de la même manière qu'un objet local :

```
bool result = chat.TransfertMessage(message);
```

Comme pour le serveur distant, la configuration peut être effectuée en utilisant la méthode statique `Configure` de la classe `RemotingConfiguration` qui prend en paramètre le nom du fichier de configuration :

```
RemotingConfiguration.Configure("client.config", false);
```

Le fichier de configuration **client.config** est au format XML et comprend toutes les informations permettant de se connecter au serveur :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown
           type="Chat, RemotingLibrary"
           url="tcp://localhost:10001/Chat"
            />
      </client>
```

```
    </application>
  </system.runtime.remoting>
</configuration>
```

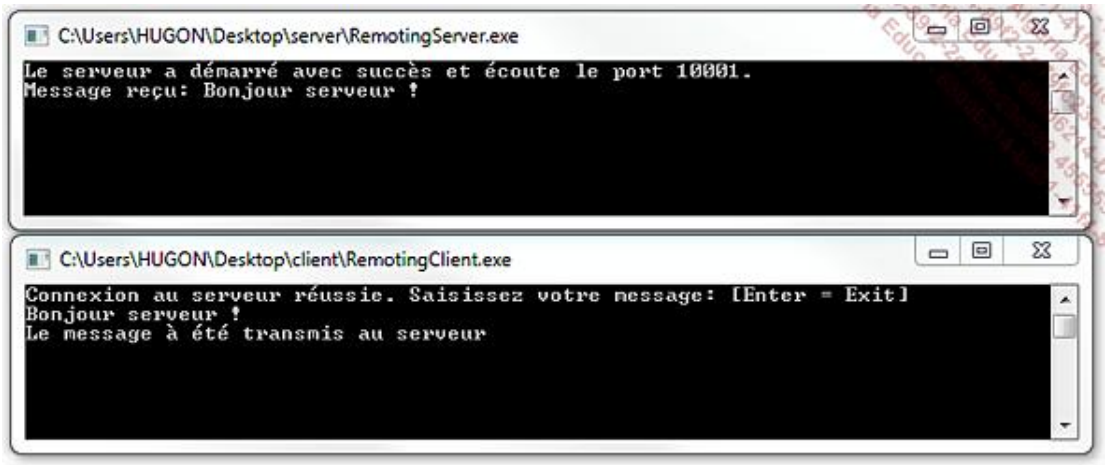
Le listing suivant détaille la classe `Program` du projet `RemotingClient` :

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingLibrary;

namespace RemotingClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                TcpChannel channel = new TcpChannel();
                ChannelServices.RegisterChannel(channel, false);
                Chat chat =
(Chat)Activator.GetObject(typeof(Chat),
"tcp://localhost:10001/Chat");

                Console.WriteLine("Connexion au serveur réussie.
Saisissez votre message: [Enter = Exit]");
                bool exit = false;
                while (!exit)
                {
                    string message = Console.ReadLine();
                    if (message != string.Empty)
                    {
                        try
                        {
                            if (chat != null)
                            {
                                bool result =
chat.TransfertMessage(message);
                                if (result)
                                {
                                    Console.WriteLine("Le message
a été transmis au serveur");
                                }
                                else
                                {
                                    throw new Exception();
                                }
                            }
                        }
                        catch
                        {
                            Console.WriteLine("Erreur de
communication avec le serveur");
                        }
                    }
                    else
                    {
                        exit = true;
                    }
                }
            }
            catch
            {
                Console.WriteLine("Erreur de connexion au
serveur");
            }
        }
    }
```

```
}
```

Pour tester le fonctionnement de ces applications, lancez d'abord l'application serveur puis l'application cliente. Saisissez un message dans l'application cliente et vérifiez qu'il est bien reçu par l'application serveur :

Il faut faire attention au firewall Windows qui pourrait bloquer les communications sur le port 10001.



La reflection

Les exemples de cette section sont disponibles dans les sources sous le projet **Reflection**.

Le Framework .NET expose dans l'espace de noms `System.Reflection` les classes permettant d'accéder aux métadonnées d'un assemblage, d'en énumérer les types et leurs membres. Cet espace de noms comprend de nombreuses classes avec, parmi les plus communes, les classes `Assembly`, `Module`, `MethodInfo`, `FieldInfo`, `PropertyInfo` ou `EventInfo`.

1. La classe System.Type

La classe `Type` de l'espace de noms `System` est au cœur du processus de réflexion, cette classe modélise les types et permet d'en connaître les détails comme leurs noms, leur espace de noms contenant ou si ce sont des types valeur ou référence.

Vous pouvez obtenir un objet `Type` à partir d'une instance d'objet, en appelant la méthode `GetType` héritée de la classe `Object` :

```
int i = 1;
Type typeInt = i.GetType();

Console.WriteLine(typeInt.FullName);
Console.WriteLine(typeInt.Namespace);
Console.WriteLine(typeInt.Name);
```

La sortie de l'exemple précédent est la suivante :

```
System.Int32
System
Int32
```

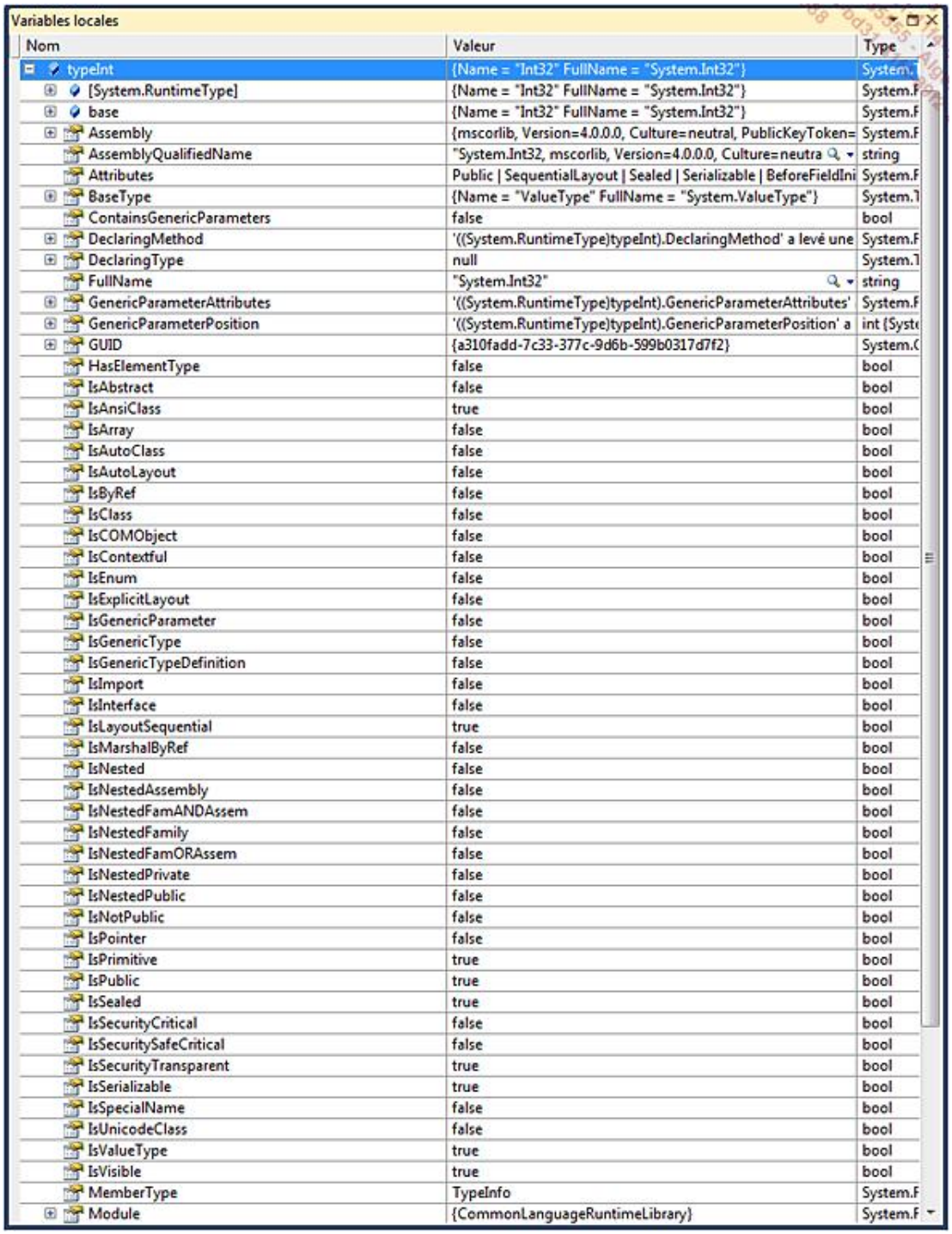
Le mot clé `typeof` permet également de récupérer un objet `Type` lié au type passé en paramètre du mot clé :

```
Type typeString = typeof(string);

Console.WriteLine(typeString.FullName);
Console.WriteLine(typeString.Namespace);
Console.WriteLine(typeString.Name);
```

La sortie de l'exemple précédent est la suivante :

```
System.String
System
String
```

La classe `Type` contient énormément de propriétés pour décrire un type. Examinez la fenêtre des **Variables locales** pour les découvrir :



2. Charger un assemblage dynamiquement

La classe abstraite `Assembly` permet de charger un assemblage à partir d'un fichier compilé, que ce soit un exécutable ou une librairie grâce à sa méthode statique `LoadFrom`. L'exemple suivant charge l'assemblage **OperatorOverloading.exe** créé précédemment :

```
Assembly assembly = Assembly.LoadFrom(
@"..\..\..\OperatorOverloading\bin\Debug\OperatorOverloading.exe"
);
```

a. L'énumération des types

À partir d'un objet de type `Assembly` et de sa méthode `GetTypes`, vous pouvez retrouver tous les types définis dans

un assemblage et en faire une énumération :

```
Type[] types = assembly.GetTypes();
foreach (Type type in types)
{
    Console.WriteLine(type.FullName);
}
```

La sortie de l'exemple précédent est la suivante :

```
OperatorOverloading.Program
OperatorOverloading.Vector
```

Lorsque le type souhaité est récupéré, les méthodes du type `Type` permettent de retrouver tous les champs avec la méthode `GetFields`, toutes les propriétés avec la méthode `GetProperties`, toutes les méthodes avec la méthode `GetMethods` ou encore tous les évènements, constructeurs, interfaces ou types liés. La méthode `GetMembers` permet quand à elle de retrouver tous les membres du type. L'exemple suivant montre comment énumérer les champs et méthodes du type `Vector` :

```
Type typeVector = assembly.GetType("OperatorOverloading.Vector");

FieldInfo[] fields = typeVector.GetFields();
foreach (FieldInfo field in fields)
{
    Console.WriteLine(field.Name);
}

MethodInfo[] methods = typeVector.GetMethods();
foreach (MethodInfo method in methods)
{
    Console.WriteLine(method.Name);
}
```

La sortie de l'exemple précédent est la suivante :

```
X
Y

ToString
Addition
op_Addition
op_Addition
op_Equality
op_Inequality
Equals
GetHashCode
GetType
```

Un champ est décrit par un objet `FieldInfo` tandis qu'une méthode sera décrite par un objet `MethodInfo`. Leur point commun est de dériver de la classe de base `MemberInfo` ce qui leur permet d'être retrouvés avec la méthode `GetMembers` de l'objet `Type` après une conversion explicite.

b. L'instanciation d'objets

Il existe deux techniques pour instancier un objet dont le type est chargé au moment de l'exécution. La première consiste à instancier un objet avec la méthode `CreateInstance` de la classe `Activator` en passant en paramètre le type souhaité :

```
object o = Activator.CreateInstance(typeVector);
// X=0 Y=0
```

La classe `Activator` permet de créer des objets localement, à distance ou d'obtenir une référence d'objet distant (comme vu dans la section **Remoting**).

La méthode `CreateInstance` crée une instance du type passé en paramètre et le retourne dans un type `object`. La raison est que le type n'est pas forcément connu au moment de la compilation, puisque, comme dans notre exemple, l'assemblage contenant le type peut être chargé au moment de l'exécution.

Si le constructeur possède des arguments, vous pouvez les passer à la méthode `CreateInstance`. Le constructeur adéquat sera choisi en fonction des paramètres et de leur type :

```
object o = Activator.CreateInstance(typeVector, 8, 9);
// X=8 Y=9
```

La seconde technique consiste à utiliser la méthode `GetConstructor` de l'objet `Type` pour récupérer le constructeur souhaité. La méthode `GetConstructor` prend en paramètre un tableau de `Type` décrivant les types de la signature du constructeur recherché :

```
ConstructorInfo construct = typeVector.GetConstructor(new Type[]
                                { typeof(int), typeof(int) });
```

Si aucun constructeur ne correspond à la signature indiquée, la valeur `null` sera retournée. Dans le cas contraire, un objet `ConstructorInfo` est retourné et l'appel de sa méthode `Invoke` en passant en paramètre un tableau d'objets contenant les paramètres du constructeur permet de récupérer une instance d'objet :

```
object o = construct.Invoke(new object[] { 5, 20 });
// X=5 Y=20
```

c. L'utilisation des membres

Lorsque votre objet est instancié, vous souhaiterez certainement utiliser ses membres, leur assigner des valeurs ou exécuter des méthodes. Pour assigner une valeur à un champ, il faut commencer par récupérer une instance d'objet `FieldInfo` du champ souhaité avec la méthode `GetField` de l'objet `Type` :

```
FieldInfo fieldX = typeVector.GetField("X");
```

Appelez ensuite la méthode `SetValue` de l'objet `FieldInfo` en passant en paramètres l'objet dont la propriété doit être assignée ainsi que la valeur :

```
fieldX.SetValue(o1, 1);
```

Si l'objet ne contient pas la propriété décrite par l'objet `FieldInfo`, une exception du type `ArgumentException` sera levée.

L'exemple suivant instancie deux objets de type `Vector` et leur assigne des valeurs aux propriétés `X` et `Y` :

```
object o1 = Activator.CreateInstance(typeVector);
object o2 = Activator.CreateInstance(typeVector);

FieldInfo fieldX = typeVector.GetField("X");
fieldX.SetValue(o1, 1);
fieldX.SetValue(o2, 2);

FieldInfo fieldY = typeVector.GetField("Y");
fieldY.SetValue(o1, 3);
fieldY.SetValue(o2, 10);

Console.WriteLine("o1: {0}", o1);
Console.WriteLine("o2: {0}", o2);
```

La sortie de l'exemple précédent est la suivante :

```
o1: X=1 Y=3
o2: X=2 Y=10
```

Pour exécuter une méthode, commencez par récupérer une instance d'objet `MethodInfo` avec la méthode `GetMethod` de l'objet `Type` :

```
MethodInfo methodAdd = typeVector.GetMethod("Addition");
```

La méthode `GetMethod` prend en paramètre le nom de la méthode à retrouver.

Appelez la méthode `Invoke` de l'objet `MethodInfo` en passant en paramètres l'objet sur lequel exécuter la méthode et un tableau d'objets correspondant aux paramètres de la méthode invoquée :

```
object o = methodAdd.Invoke(o1, new object[] { o2 });
Console.WriteLine("o (o1 + o2): {0}", o);
```

La sortie de l'exemple précédent est la suivante :

```
o (o1 + o2): X=3 Y=13
```

Introduction

Un assemblage représente la brique de base d'une application. Il contient la description des types et facilite la réutilisation du code en formant une unité pouvant être facilement référencée dans un autre assemblage. Les assemblages sont auto descriptifs. Ils contiennent toutes les informations utiles pour leur interprétation et configuration.

Les assemblages privés

Tous les assemblages qui ont été créés au cours des exemples de l'ouvrage sont des assemblages privés. C'est le type d'assemblage créé par défaut par Visual Studio. Un assemblage privé est un assemblage qui ne peut être utilisé que par une seule application. Il est ou fait partie intégrante de l'application. Pour schématiser, un assemblage est un projet qui se compile en un fichier avec l'extension **.exe** ou **.dll**. Si un projet contient une référence à un autre assemblage, lors de la compilation, il y aura copie de l'assemblage ainsi lors de la modification de l'assemblage référencé, il faudra compiler à nouveau le projet pour que la nouvelle version de l'assemblage soit prise en compte. Il n'y a donc pas le partage d'un même assemblage.

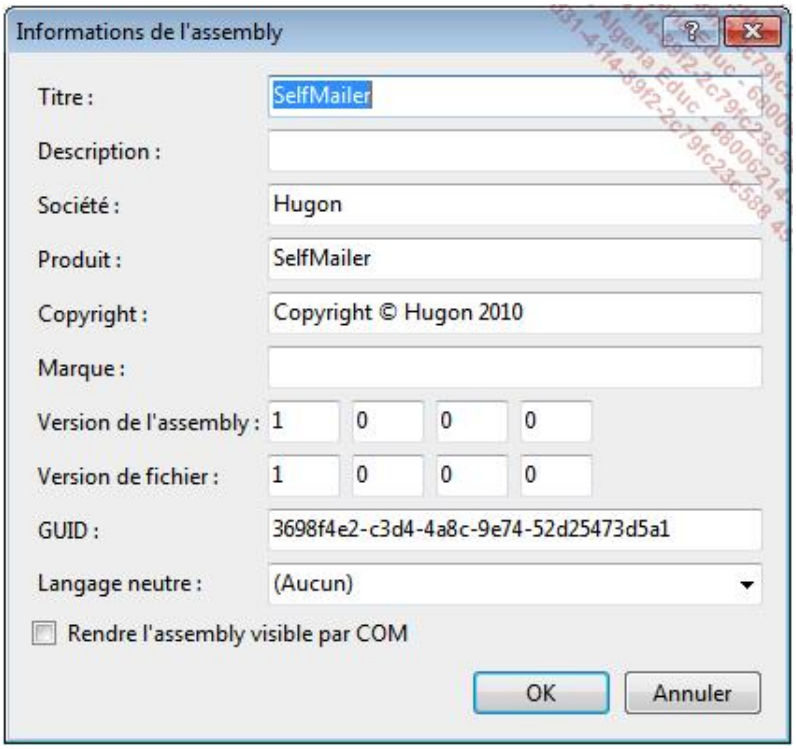
Un assemblage est découpé en quatre parties :

- Le manifeste : il contient les informations sur l'assemblage comme son identité, les types, les ressources, les fichiers ou les permissions de sécurité.
- Les métadonnées de type : elles contiennent la description des types inclus dans l'assemblage.
- Le code IL : les instructions de l'assemblage sous forme de langage intermédiaire.
- Les ressources : elles contiennent tous les contenus correspondant à une culture spécifique, il peut ne pas y en avoir.

Pour la majorité des informations du manifeste, Visual Studio s'en charge au moment de la compilation. Il reste néanmoins à la charge du développeur de définir l'identité de l'assemblage. L'identité d'un assemblage est caractérisée par un nom, un numéro de version, une description, un copyright et bien d'autres. Toutes ces informations sont saisies, par défaut, dans le fichier **AssemblyInfo.cs** dans le dossier **Properties** d'un projet :

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("SelfMailer")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Hugon")]
[assembly: AssemblyProduct("SelfMailer")]
[assembly: AssemblyCopyright("Copyright © Hugon 2010")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("3698f4e2-c3d4-4a8c-9e74-52d25473d5a1")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Les informations de l'assemblage sont également éditables depuis la page des propriétés du projet, sous l'onglet **Application** en ouvrant la fenêtre **Informations de l'assembly** :



La plupart des applications Windows utilisent des ressources notamment pour les images, les icônes et les éléments culturels. Toutes ces données ne sont pas exécutables. Elles sont juste rassemblées de manière à être utilisées plus facilement par l'application. Pour localiser une application, vous ajoutez différents fichiers de ressources en fonction des cultures qui doivent être couvertes par l'application comme vu dans le chapitre Globalisation et Localisation. Si vous observez la génération du projet **SelfMailer**, vous remarquerez que le dossier **bin\Debug** contient un dossier supplémentaire **en-US** avec une librairie nommée **SelfMailer.resources.dll**. Cet assemblage est un assemblage satellite de l'application **SelfMailer**.

Les assemblages satellites sont chargés automatiquement lors de l'exécution de l'application en fonction des paramètres culturels de la machine.

Les assemblages partagés

Les assemblages peuvent être privés ou partagés. Alors qu'un assemblage privé n'est utilisé que par une seule application, un assemblage partagé pourra être utilisé par plusieurs applications car une seule copie est présente sur la machine. Pour être partagé, un assemblage doit être installé dans le GAC (*Global Assembly Cache*). C'est le cas des librairies que composent le Framework .NET et c'est la raison pour laquelle aucune de ces librairies n'est copiée lors de la compilation du projet.

Le partage d'un assemblage est avantageux si plusieurs applications ont besoin d'accéder à la même copie d'un assemblage mais aussi parce que le GAC se trouve dans un dossier Windows qui bénéficie du niveau de sécurité le plus élevé. Un autre atout des assemblages partagés est de pouvoir installer plusieurs versions du même assemblage dans le GAC. Les applications pourront alors référencer et utiliser la version appropriée.

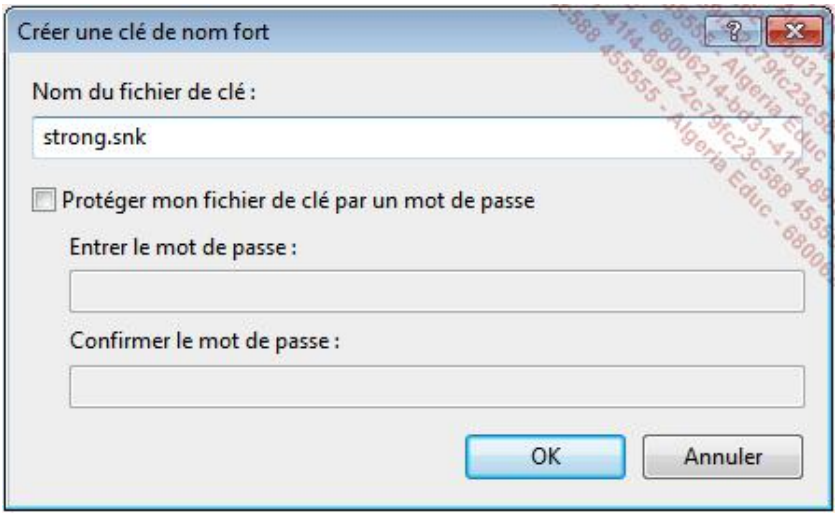
Pour installer un assemblage dans le GAC, il doit être signé avec un nom fort. Ce nom garantit l'identité de l'assemblage. Il est composé de son nom, de sa version, de ses informations de culture s'il y a besoin. Ces informations sont encryptées avec un algorithme asymétrique à l'aide d'une clé privée et peuvent être décryptées avec une clé publique. Comme le développeur est le seul à posséder la clé privée, le nom fort ne pourra pas être répliqué ce qui certifie l'identité de l'assemblage.

La première chose à faire pour signer un assemblage est d'obtenir une paire de clés privée/publique. Vous pouvez en générer une à partir de l'outil sn.exe fourni avec Visual Studio :

- Ouvrez une **Invite de commande de Visual Studio 2010** à partir du menu **Démarrer** - **Tous les programmes** - **Microsoft Visual Studio 2010** - **Visual Studio Tools**.
- Saisissez `sn` dans l'invite de commande pour obtenir l'aide et la description de l'outil.
- Pour générer une paire de clés, saisissez `sn -k fichier.snk`. La paire sera générée dans le fichier spécifié dont l'extension est généralement .snk.

Visual Studio met à disposition une interface plus élégante pour générer une paire de clés :

- Ouvrez la fenêtre des propriétés d'un projet et allez sur l'onglet **Signature**.
- Cochez la case **Signer l'assembly** et dans la liste déroulante du choix d'un fichier de clé, sélectionnez **<Nouveau...>**.
- La fenêtre **Créer une clé de nom fort** s'ouvre. Saisissez le nom de votre fichier de clé et vous pouvez spécifier un mot de passe pour protéger le fichier généré :



- Après validation, le fichier est créé à la racine du projet.

Lorsque la clé est associée au projet dont l'option **Signer l'assembly** est cochée, il suffit de générer le projet pour que le nom fort soit généré et signé dans l'assemblage. Pour installer l'assemblage dans le GAC, utilisez l'utilitaire **gacutil.exe** avec l'option `/i` et le chemin de l'assemblage pour spécifier qu'il doit être installé :

```
gacutil /i assemblage.dll
```

Les fichiers de configuration

Lorsqu'un assemblage est compilé et déployé, rien n'est plus modifiable sans mise à jour. Les fichiers de configuration permettent de configurer une application dans un fichier au format XML sans avoir à la recompiler. Ce type de fichiers se nomme en général **App.config** dans un projet puis ils prennent le nom de l'assemblage avec l'extension **.config** après compilation. Pour exemple, le projet **SelfMailer** qui contient un fichier **App.config** devient le fichier **SelfMailer.exe.config** après compilation.

La structure du fichier de configuration doit répondre à un schéma spécifique. Sa structure de base minimale est la suivante :

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
</configuration>
```

Visual Studio ajoute automatiquement un fichier de configuration si des informations, comme les chaînes de connexion SQL, doivent y être stockées et que le projet n'en contient pas. Pour ajouter un fichier de configuration manuellement, ouvrez la fenêtre **Ajouter un nouvel élément** et sélectionnez **Fichier de configuration de l'application**.

Toutes les sections du fichier de configuration ne seront pas abordées dans cet ouvrage. Nous allons nous concentrer sur la définition de paramètres personnalisés et sur la manière de les utiliser.

L'IntelliSense de Visual Studio permet d'éditer plus facilement le fichier de configuration. Ajoutez une balise `appSettings` sous la balise racine et définissez le paramètre `AppTitle` suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <appSettings>
           <add key="AppTitle" value="Titre configuré"/>
    </appSettings>
    ...
</configuration>
```

La section `appSettings` du fichier de configuration permet de spécifier des paires clé/valeur de paramètres qui seront chargées au moment de l'exécution. La valeur peut alors être retrouvée à tout moment de l'application avec la classe `ConfigurationManager` de l'espace de noms `System.Configuration`. Pour utiliser cette classe, vous devez ajouter une référence à l'assemblage **System.configuration** à votre projet. Vous pouvez ensuite accéder aux paramètres définis dans le fichier de configuration de la manière suivante :

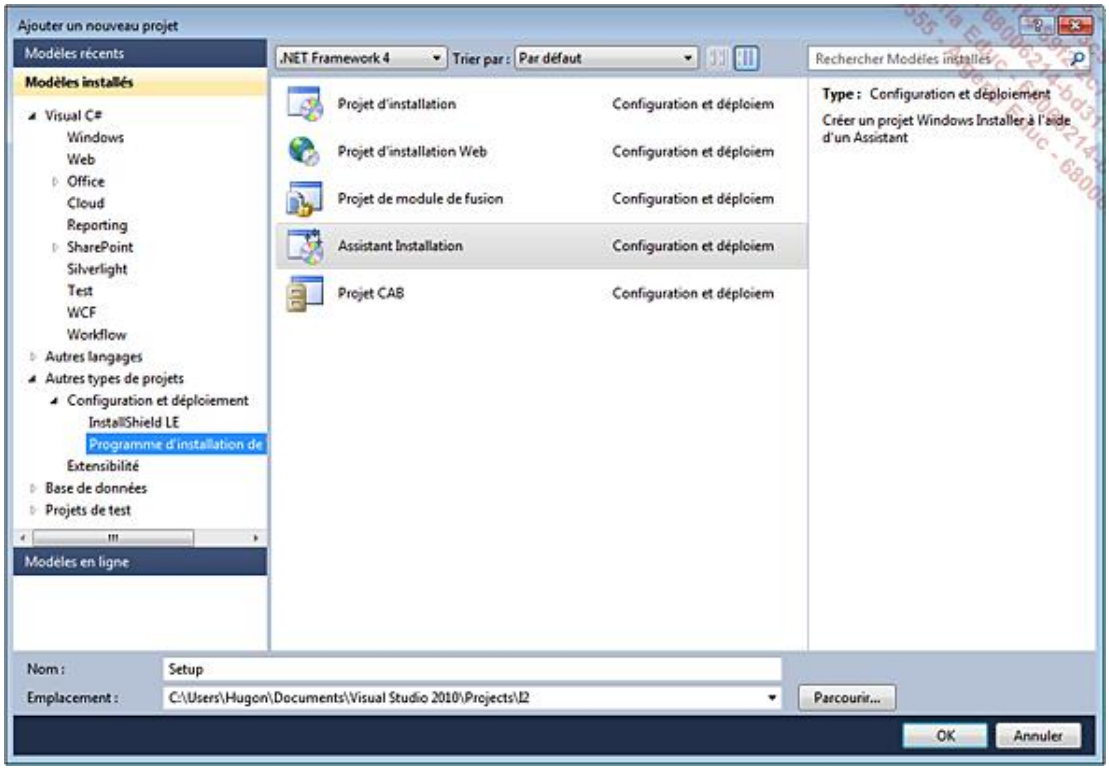
```
string AppTitle = ConfigurationManager.AppSettings["AppTitle"];
```

La classe `ConfigurationManager` expose la collection des paramètres du fichier de configuration dans une propriété `AppSettings`. Vous pouvez accéder à la valeur d'un paramètre soit par sa clé soit par son index.

Le fichier de configuration est lu une seule fois dans le cycle de vie d'une application, à son lancement. Toute modification du fichier de configuration est prise en compte seulement si l'application est redémarrée.

Introduction

Un projet de déploiement est la dernière phase dans la conception d'une application. Il s'agit de fournir tous les éléments d'une application d'une manière à ce qu'elle soit facilement distribuable. Visual Studio propose des modèles de projet d'installation dans la fenêtre **Ajouter un nouveau projet**, sous la catégorie **Autres types de projets** - **Configuration et déploiement** - **Programme d'installation de Visual Studio** :



Les projets de déploiement

1. XCOPY

Le déploiement XCOPY n'est pas un modèle de projet disponible dans Visual Studio. Ce type de déploiement porte le nom de la commande DOS utilisée pour le déploiement. En effet, ce type de déploiement utilise la commande XCOPY pour copier le contenu d'un répertoire vers un autre répertoire cible. Il s'agit de la méthode la plus simple et rapide mais aussi de la plus limitée pour déployer une application.

Pour être déployée avec cette méthode, le dossier de l'application doit contenir d'une part, tous les fichiers requis à son bon fonctionnement et d'autre part, le Framework .NET doit être installé sur la machine cible.

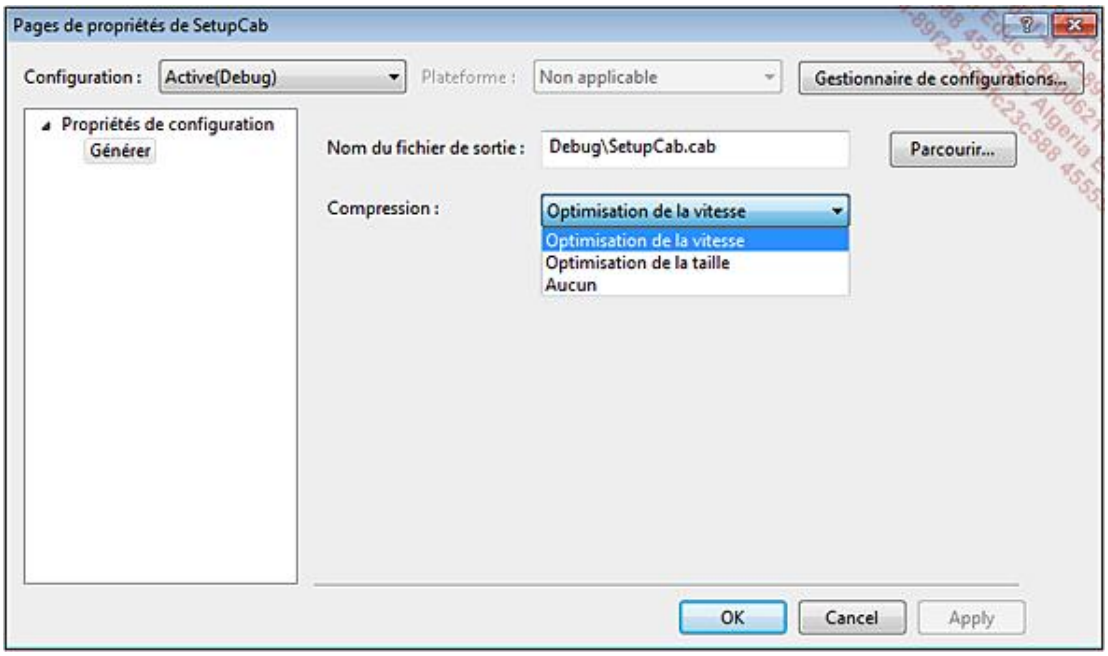
Pour réaliser un déploiement XCOPY, ouvrez une invite de commande et exécutez la commande suivante :

```
XCOPY C:\DossierSource C:\DossierDestination /s
```

L'option `/s` spécifie que les sous-répertoires doivent également être copiés.

2. Le projet CAB

Un projet CAB a pour vocation de créer des fichiers **.cab** pour une distribution à partir d'anciennes technologies de déploiement. Vous pouvez ajouter la sortie des projets ou des fichiers dans le projet CAB. Tous ces fichiers seront empaquetés dans le fichier spécifié dans les options du projet avec soit une optimisation de la taille, soit une optimisation de la vitesse ou encore aucune optimisation :



3. Le projet de module de fusion

Les projets de module de fusion sont principalement utilisés pour configurer le déploiement de composants et de librairies qui sont utilisés dans plusieurs projets. Le projet de module de fusion contient toute la configuration de l'installation du composant et génère un fichier avec l'extension **.msm**.

Lorsque le composant est utilisé par une application, au lieu de l'ajouter au projet de déploiement de l'application et de refaire la configuration de son déploiement, vous pouvez directement ajouter le projet de module de fusion.

4. Le projet d'installation

Les projets d'installation créent des programmes d'installation dans un fichier Windows Installer ayant

l'extension **.msi**. L'utilisation de Windows Installer présente de nombreux avantages pour le déploiement d'application :

- Rollback des opérations effectuées en cas d'erreur lors de l'installation.
- Configuration de pré-requis à l'installation.
- L'application apparaît dans la fenêtre **Ajouter ou supprimer des programmes** de Windows et peut être désinstallée ou réparée à partir de cet emplacement.

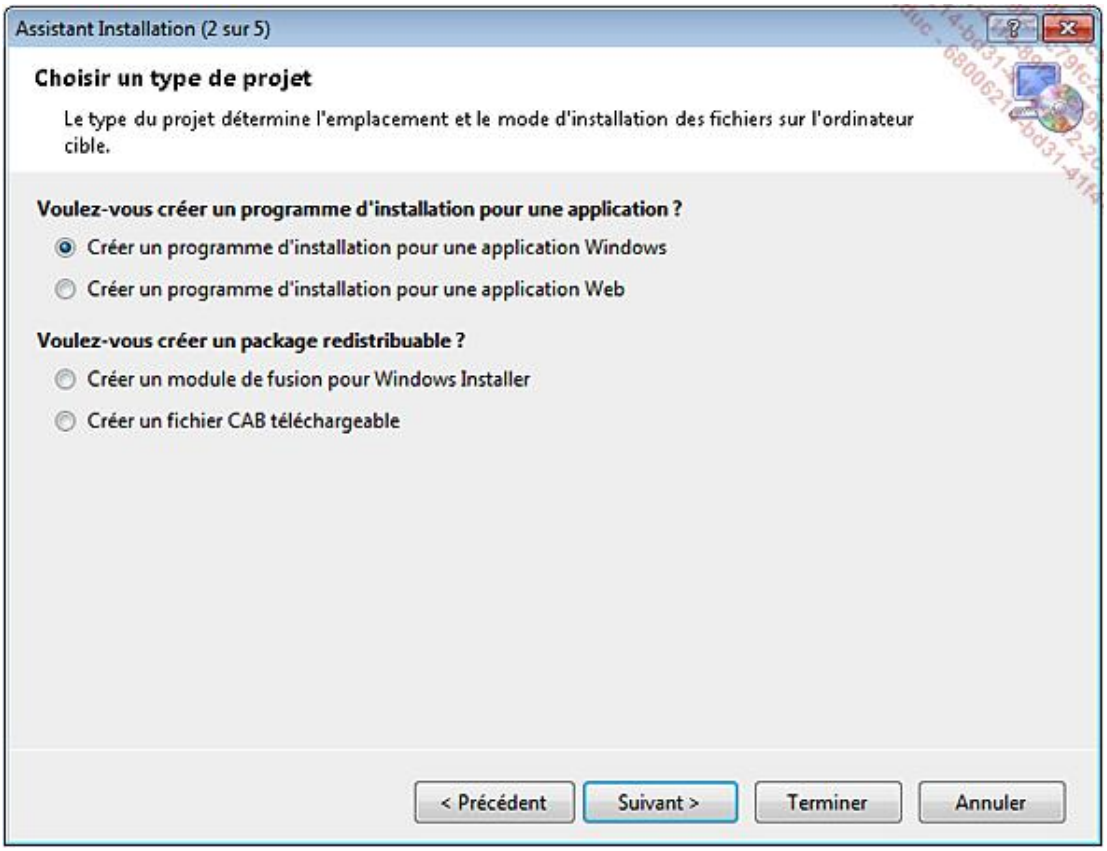
Il existe deux types de projets d'installation : les projets d'installation Web qui sont utilisés pour le déploiement d'applications Web (les fichiers sont déployés sur un serveur Web IIS). Les projets d'installation Windows qui sont utilisés pour le déploiement d'applications Windows (les fichiers sont déployés sur l'ordinateur cible).

Vous ne pouvez pas modifier le type de projet après sa création, un projet d'installation Windows n'est pas modifiable en projet d'installation Web.

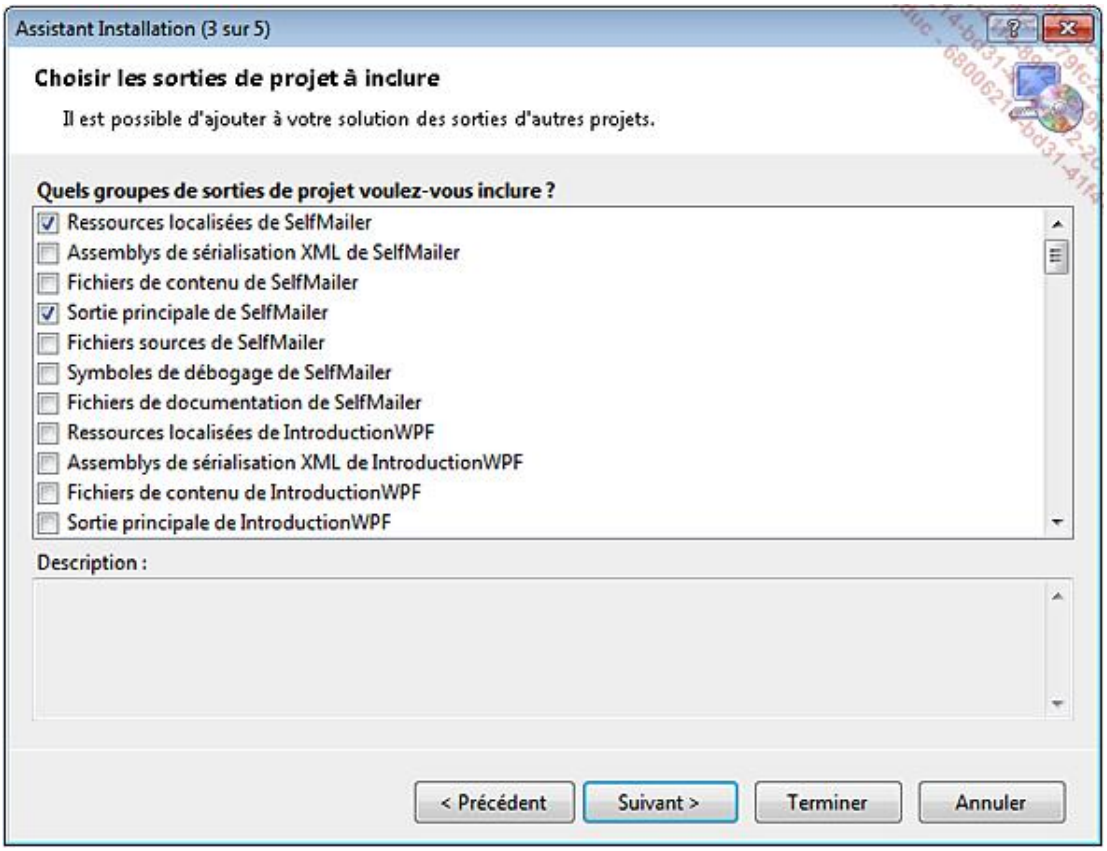
L’assistant Installation

Visual Studio propose l’assistant Installation permettant de créer rapidement les bases d’un projet d’installation tout en étant guidé.

Ajoutez un nouveau projet à la solution **SelfMailer** nommé **Setup** en sélectionnant l’assistant Installation. L’assistant est composé de 5 étapes. La première étape est simplement un message de bienvenue. Cliquez sur le bouton **Suivant** pour afficher la seconde étape :



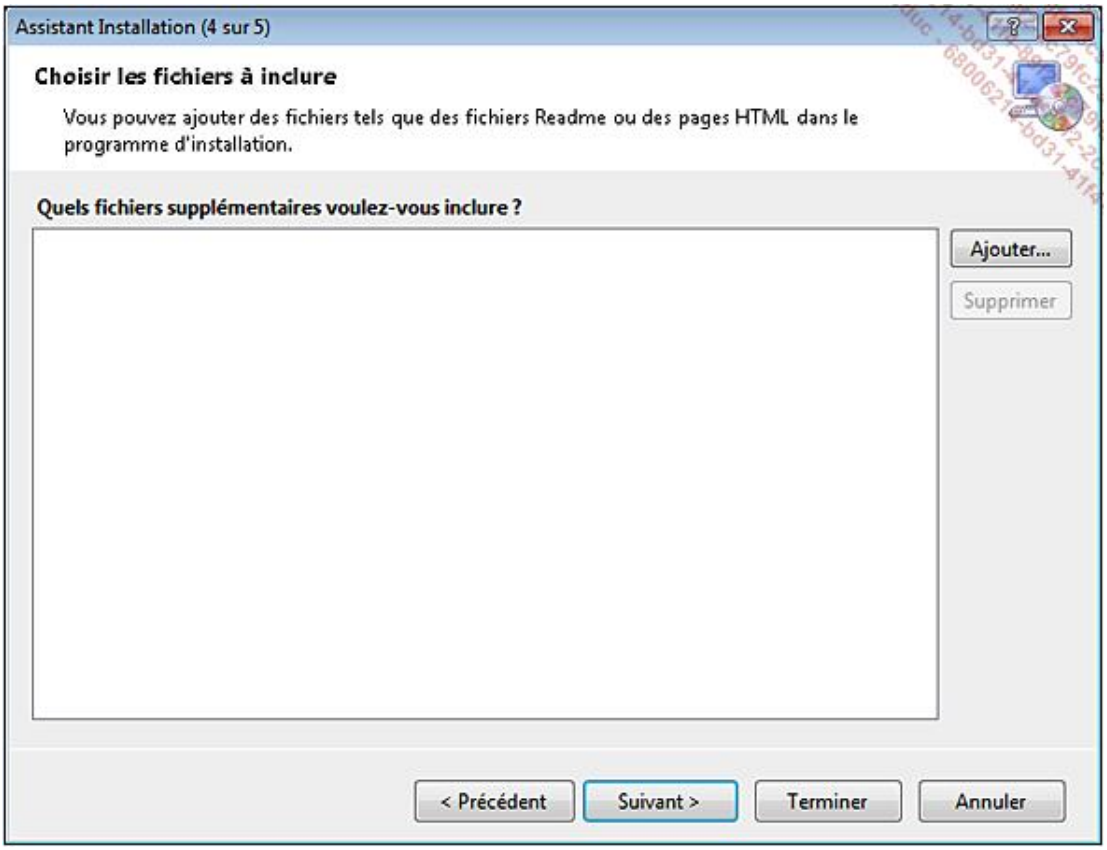
L’étape 2 vous permet de sélectionner le type de projet de déploiement que vous souhaitez créer. Sélectionnez **Créer un programme d’installation pour une application Windows** et cliquez sur le bouton **Suivant** pour passer à l’étape 3 :



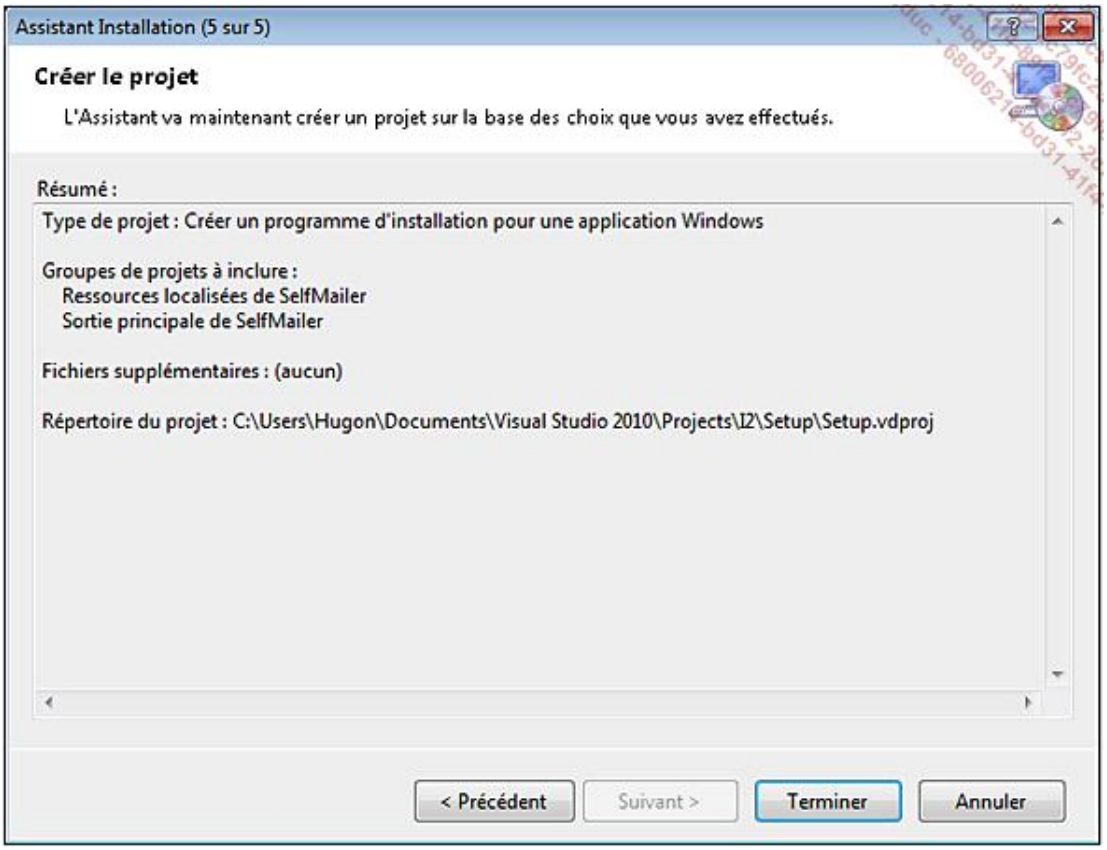
L'étape 3 a pour but de sélectionner les sorties de projet à inclure. Tous les projets de la solution courante sont disponibles. Pour chacun d'eux, sept types de sorties sont proposées :

- Les ressources localisées contiennent les assemblages satellites. Sélectionnez ce groupe pour une application localisée.
- Les assemblages de sérialisation XML contiennent les assemblages utiles à la sérialisation XML permettant de ne pas passer par un processus de reflection pour la lecture et l'écriture des éléments XML.
- Les fichiers de contenu incluent tous les fichiers du projet qui ne sont pas compilés.
- La sortie principale correspond aux fichiers exécutables (**.exe** ou **.dll**) produits par la compilation du projet.
- Les fichiers sources permettent d'inclure dans le déploiement tous les fichiers de code du projet.
- Les symboles de débogage peuvent être utiles dans le projet d'installation s'il s'agit d'une phase de test. Par contre pour un déploiement définitif, cela n'a aucun intérêt de les inclure.
- Les fichiers de documentation représentent la documentation XML produite au fil du développement. L'inclusion de ces fichiers est utile dans le cas du déploiement d'une librairie.

Cochez les cases **Ressources localisées de SelfMailer** et **Sortie principale de SelfMailer** puis cliquez sur le bouton **Suivant** pour passer à l'étape 4 :

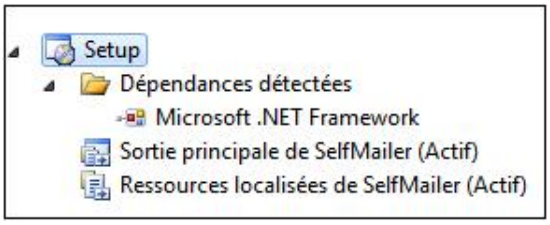


L'étape 4 permet de sélectionner des fichiers supplémentaires à inclure dans le projet de déploiement. Cela peut, par exemple, être des fichiers d'aide, un manuel d'utilisation ou des fichiers d'exemples. Une fois les fichiers supplémentaires ajoutés, cliquez sur le bouton **Suivant** pour passer à l'étape 5 :



L'étape 5 présente un récapitulatif des choix effectués lors des étapes de l'assistant. Cliquez sur le bouton **Terminer** pour finaliser la création du projet. Visual Studio crée le nouveau projet avec les sorties spécifiées et les éventuels fichiers supplémentaires. De plus, Visual Studio détecte les dépendances liées au projet et les inclus dans le dossier

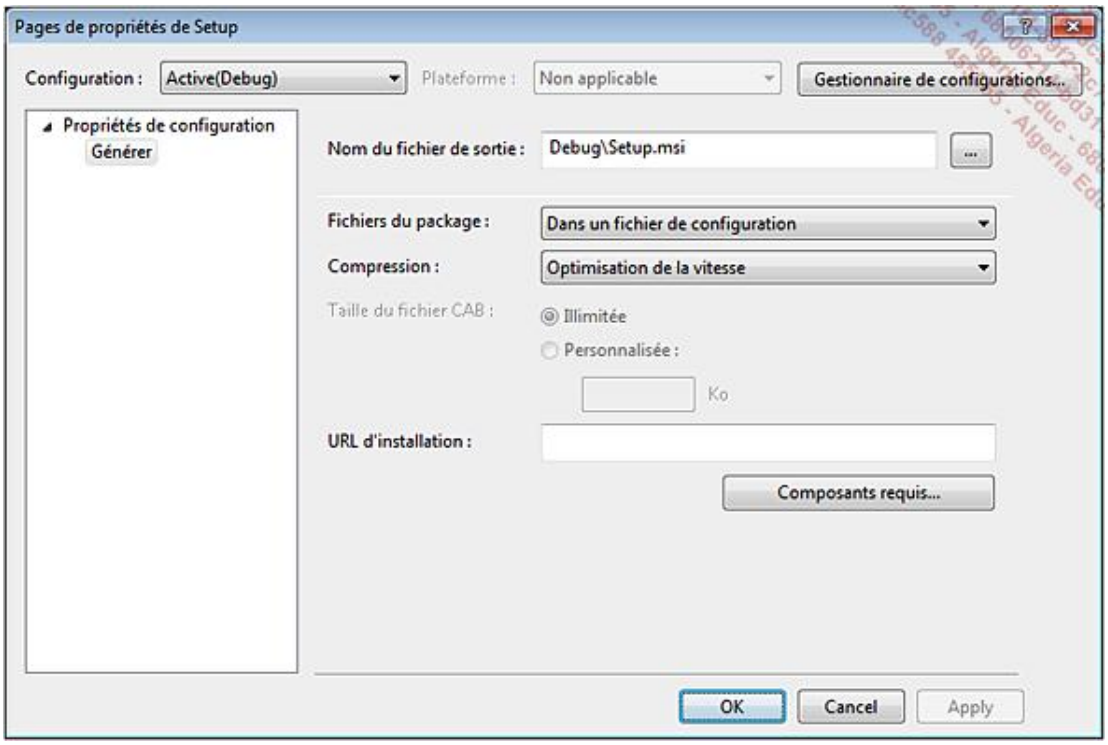
Dépendances détectées dans la fenêtre **Explorateur de solutions** :



La configuration du projet

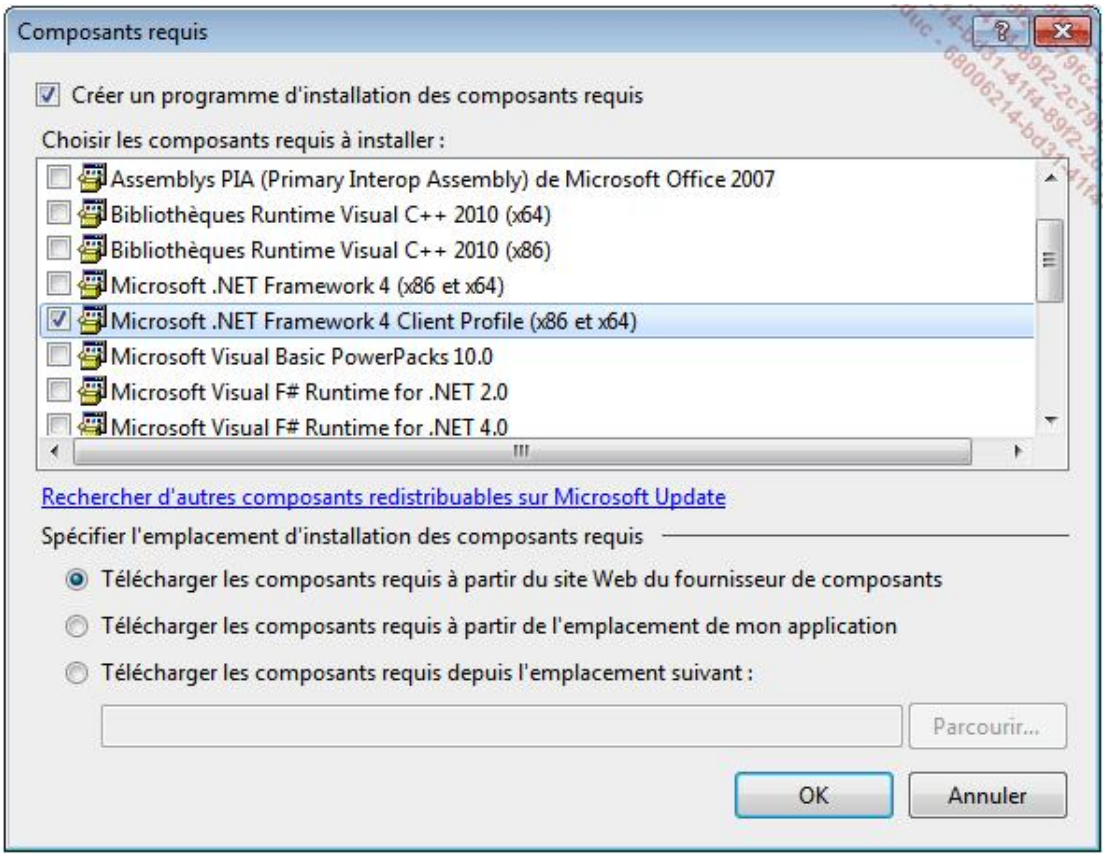
1. Les propriétés du projet

La définition des propriétés d'un projet d'installation se fait à deux endroits. Le premier est la fenêtre de propriétés du projet, accessible à partir du menu **Projet** - **Propriétés** :



Depuis cette fenêtre, vous pouvez définir les éléments suivants :

- **Nom du fichier de sortie** : ce champ permet de définir où sera généré le fichier d'installation et quel sera son nom.
- **Fichiers du package** : cette liste permet de définir si les fichiers seront inclus dans un fichier de configuration (.msi) ou dans un fichier CAB (.cab).
- **Compression** : cette option permet de spécifier le mode d'optimisation du package, soit pour la vitesse, soit pour la taille ou sans optimisation.
- **Taille du fichier CAB** : ce groupe de champs permet de définir la taille du ou des fichiers CAB qui seront produits. Le fichier produit pourra être de taille illimitée ou d'une taille personnalisée à définir.
- **Composants requis** : ce bouton ouvre la boîte de dialogue **Composants requis** permettant de sélectionner les composants dont votre application à besoin et de sélectionner quel sera le mode d'installation de ceux-ci :



Vous remarquez que le composant **Microsoft .NET Framework 4 Client Profile (x86 et x64)** est déjà coché. Son mode d'installation est le téléchargement du composant à partir du site Web du fournisseur. Les autres options d'installation d'un composant requis sont le téléchargement à partir du même emplacement que l'application ou la possibilité de spécifier un fichier local, réseau, une URL locale ou une URL distante.

La seconde partie des propriétés est accessible via la fenêtre **Propriétés** de Visual Studio. Vous pouvez configurer les propriétés suivantes :

- **AddRemoveProgramsIcon** : spécifie l'icône qui sera affichée dans la fenêtre **Ajouter ou supprimer des programmes**.
- **Author** : spécifie le nom de l'auteur de l'application.
- **Description** : spécifie une description de l'application.
- **DetectNewerInstalledVersion** : cette valeur booléenne permet de vérifier si une version plus récente de l'application est déjà installée sur la machine cible. Si c'est le cas, l'installation sera stoppée et un message indiquera qu'une version plus récente est déjà installée sur la machine.
- **InstallAllUsers** : cette propriété permet de spécifier si l'installation sera faite pour tous les utilisateurs ou seulement l'utilisateur courant.
- **Keywords** : spécifie les mots clés permettant de faire une recherche sur les programmes d'installation.
- **Localization** : cette propriété spécifie les paramètres culturels pour l'interface utilisateur.
- **Manufacturer** : spécifie le nom du fabricant de l'application.
- **ManufacturerUrl** : spécifie l'URL du site Web du fabricant de l'application.
- **PostBuildEvent** : cette propriété permet de définir les commandes qui seront exécutées après la génération du projet.

- **PreBuildEvent** : cette propriété permet de définir les commandes qui seront exécutées avant la génération du projet.
- **ProductCode** : ce GUID identifie de manière unique l'application.
- **ProductName** : spécifie le nom de l'application.
- **RemovePreviousVersions** : cette valeur booléenne permet de spécifier qu'un programme d'installation doit supprimer toutes les anciennes versions d'une application avant d'installer la nouvelle.
- **RunPostBuildEvent** : cette propriété détermine la condition qui permet d'exécuter les commandes définies dans la propriété **PostBuildEvent**.
- **SearchPath** : spécifie le chemin pour rechercher des assemblages, fichiers ou modules de fusion sur l'ordinateur de développement.
- **Subject** : spécifie une information supplémentaire de description de l'application.
- **SupportPhone**: Spécifie le numéro de téléphone du support technique de l'application.
- **SupportUrl** : spécifie l'URL du support technique de l'application.
- **TargetPlatform** : cette propriété spécifie quel est le type de plateforme ciblée. x86, x64 ou Itanium.
- **Title** : cette propriété spécifie le titre du programme d'installation.
- **UpgradeCode** : ce GUID est un identifiant représentant la version d'une application.
- **Version** : cette propriété spécifie la version du programme d'installation.

2. Les éditeurs de configuration

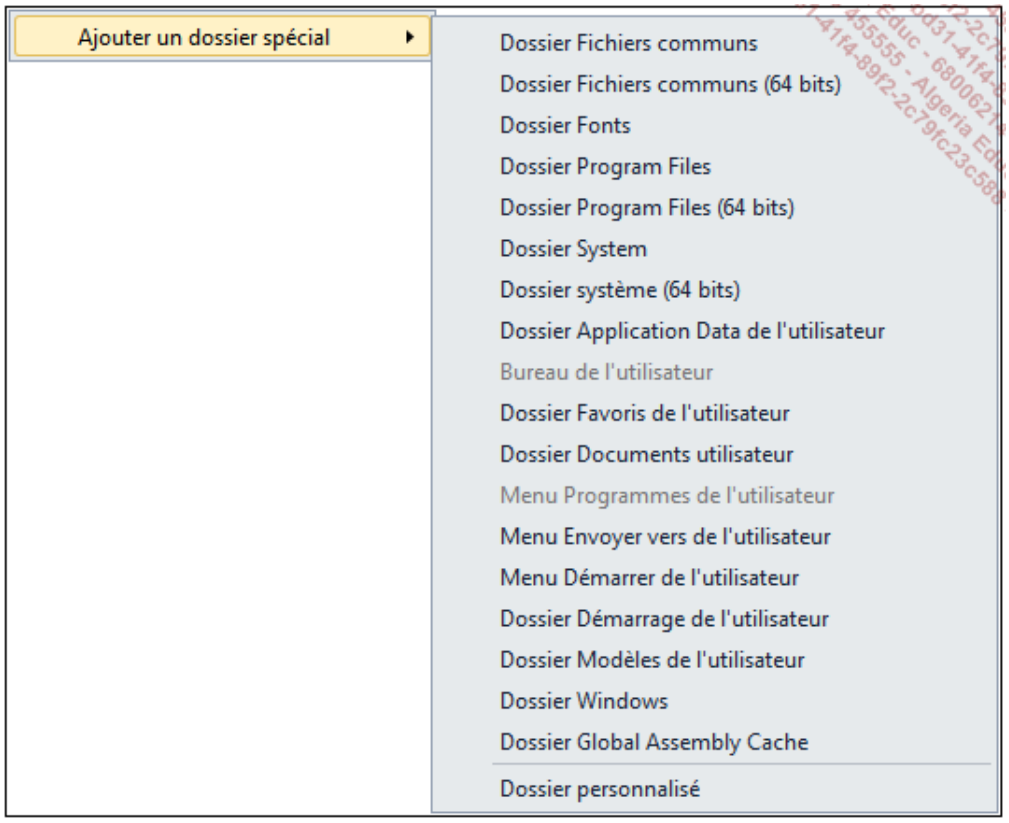
Visual Studio met à disposition six éditeurs pour la personnalisation de tous les aspects du déploiement de votre application. Ces éditeurs sont accessibles à partir du menu **Affichage** - **Editeur** ou via les boutons situés en haut de la fenêtre **Explorateur de solutions** lorsque le focus est sur un projet de déploiement :

Dans l'ordre de gauche à droite : la fenêtre des propriétés, l'éditeur du système de fichiers, l'éditeur du registre, l'éditeur des types de fichiers, l'éditeur de l'interface utilisateur, l'éditeur des actions personnalisées et l'éditeur des conditions de lancement.

a. L'éditeur du système de fichiers

L'éditeur du système de fichiers permet de configurer l'installation des fichiers de l'application sur l'ordinateur cible :

L'éditeur est composé de deux parties : à gauche est listée l'arborescence des répertoires de la machine cible et à droite les fichiers qui y seront créés. Par défaut, l'éditeur comporte trois dossiers cibles : **Bureau de l'utilisateur**, **Dossier d'application** et **Menu Programmes de l'utilisateur**. Les noms de ces dossiers sont auto descriptifs et représentent chacun une partie de la machine cible. D'autres dossiers cibles sont disponibles. Vous pouvez en ajouter avec le menu contextuel :



Pour créer un raccourci vers l'application sur le bureau de l'ordinateur cible, ouvrez le menu contextuel de la sortie principale et choisissez le menu **Créer un raccourci vers Sortie principale de SelfMailer (Actif)** :

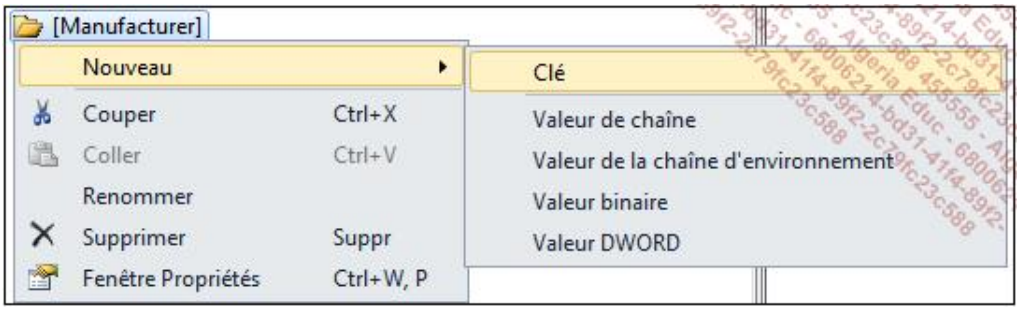
Renommez le raccourci en **SelfMailer** et déplacez-le en le faisant glisser vers le dossier **Bureau de l'utilisateur**. Le raccourci est maintenant configuré. Lors de l'installation, un raccourci de l'application sera créé sur le bureau de la machine cible.

b. L'éditeur du registre

L'éditeur de registre permet d'ajouter des entrées dans le registre de la machine cible pendant la phase d'installation de l'application :

L'éditeur est divisé en deux parties : à gauche se trouve une représentation du registre de la machine cible et à droite, les valeurs qui seront inscrites.

Pour ajouter une nouvelle clé, ouvrez le menu contextuel sur la clé parente et sélectionnez le menu **Nouveau** - **Clé**. Pour ajouter une valeur à une clé, faites la même opération en sélectionnant le type de valeur que vous souhaitez créer : une **valeur de chaîne**, une **valeur de la chaîne d'environnement**, une **valeur binaire** ou une **valeur DWORD** :

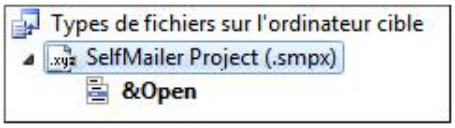


Le programme d'installation se chargera de créer les clés de registre pendant la phase d'installation mais aussi de les supprimer lors de la phase de désinstallation.

c. L'éditeur des types de fichiers

L'éditeur des types de fichiers vous permet de réaliser des associations entre des fichiers et l'application.

Créez une nouvelle association en ouvrant le menu contextuel de l'élément **Types de fichiers sur l'ordinateur cible** et en sélectionnant le menu **Ajouter un type de fichier**. Saisissez **SelfMailer Project** et dans la fenêtre propriétés, spécifiez l'extension **.smpx** dans la propriété **Extensions**. Pour associer plusieurs types de fichiers, il faut séparer les extensions avec des points virgules. Le point devant l'extension est facultatif :



La propriété **Command** permet de spécifier quel est le fichier qui sera exécuté lorsqu'une action sera lancée.

Les actions sont associées au type de fichier. Par défaut il y en a une : **&Open**. Ouvrez les propriétés de l'action. Il y en a trois : (**Name**) va définir le nom de l'action, c'est aussi ce qui sera affiché sur le menu contextuel du type de fichier. La propriété **verb** définit l'action qui sera effectuée pour ce type de fichier, en l'occurrence le fichier sera ouvert avec l'application. La propriété **Arguments** permet de spécifier les arguments qui seront adressés à l'application.

Le fait d'avoir ajouté une action d'ouverture sur un type de fichier entraîne le lancement de l'application associée mais la logique du chargement du fichier dans l'application doit être faite par le développeur. Le chemin du fichier est toujours le premier argument passé à l'application. Ouvrez le fichier **Program.cs** du projet **SelfMailer** et ajoutez le code suivant à la méthode `Main` :

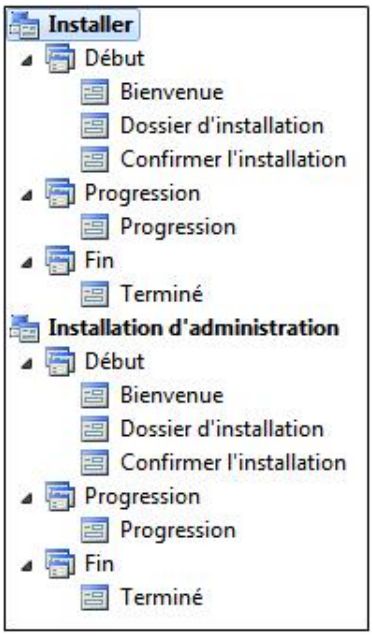
```
[STAThread]
static void Main(string[] args)
{
    // ...

    Project = new Library.Project();
    if (args.Length == 1 &&
        File.Exists(args[0]) &&
        Path.GetExtension(args[0]) == ".smpx")
    {
        Project = Library.Project.Load(args[0]);
    }

    // ...
}
```

La signature de la méthode `Main` doit être modifiée pour accepter un tableau de type `string` en argument.

Cette portion de code détecte si l'application a été lancée avec un argument et si cet argument représente un fichier de projet, il est automatiquement chargé dans l'interface.

d. L'éditeur de l'interface utilisateur

L'éditeur de l'interface utilisateur permet de modifier les boîtes de dialogue qui constitueront l'assistant d'installation de l'application. Il existe deux types d'interface utilisateur par défaut : **Installer** et **Installation d'administration** :

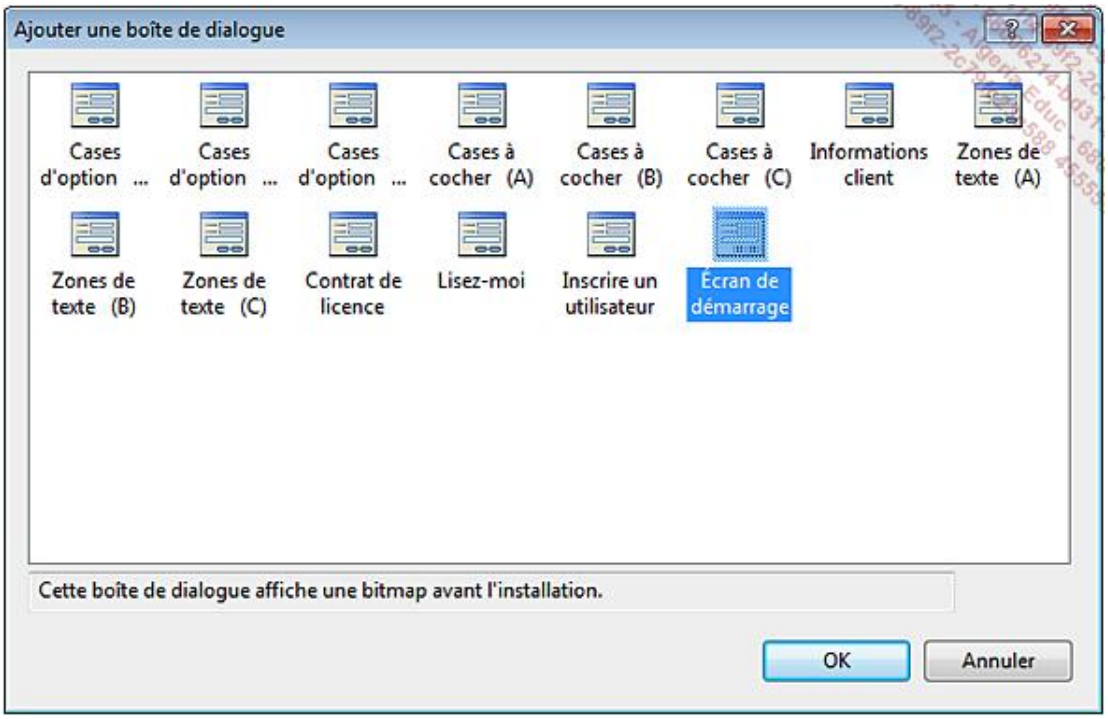


L'installation administrative d'une application permet de la déployer depuis un partage réseau. Une image de l'application et du fichier d'installation est alors disponible aux utilisateurs. L'administrateur aura préalablement choisi des options d'installation comme par exemple le dossier d'installation par défaut et pour qu'il ne soit pas modifié par un utilisateur, la boîte de dialogue **Dossier d'installation** sera supprimée de la procédure d'installation normale.

Chaque installation est découpée en trois phases contenant une ou plusieurs boîtes de dialogue : **Début**, **Progression** et **Fin**. La phase **Début** permet de recueillir les informations de l'utilisateur, le dossier d'installation ou encore la présentation du contrat de licence. À la fin de cette première phase, Windows Installer effectue les vérifications concernant notamment l'espace disque et les pré-requis avant de lancer la phase de **Progression**. Cette seconde phase ne contient qu'une boîte de dialogue car il n'y a pas besoin de pouvoir interagir avec l'utilisateur. Le but de cette boîte de dialogue est de présenter la progression et que l'utilisateur puisse annuler l'installation. La dernière phase, **Fin**, est lancée dès que la seconde phase est terminée, elle permet de présenter des informations sur l'installation ou proposer des options supplémentaires à l'utilisateur comme l'ajout de raccourcis.

Chaque boîte de dialogue possède une série de propriétés qui lui sont propres. Par exemple, la boîte **Bienvenue** comprend trois propriétés : **BannerBitmap** pour spécifier une image à afficher dans la bannière de la boîte de dialogue, **CopyrightWarning** correspondant à un texte d'avertissement affiché en bas de la boîte et **WelcomeText** qui permet de définir le texte de bienvenue en haut de la boîte de dialogue. Toutes les boîtes de dialogue ont en commun la propriété **BannerBitmap**. L'ordre des boîtes de dialogue est modifiable en réalisant un glissé-déposé.

Vous pouvez ajouter une nouvelle boîte de dialogue en ouvrant le menu contextuel de la phase à laquelle vous souhaitez ajouter l'élément et en sélectionnant le menu **Ajouter une boîte de dialogue**. La fenêtre **Ajouter une boîte de dialogue** s'affiche, vous présentant un choix d'éléments :

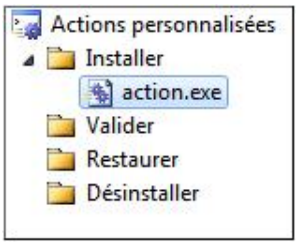


Sélectionnez le type de boîte de dialogue désiré et cliquez sur le bouton **OK** pour l'ajouter au projet de déploiement.

e. L'éditeur des actions personnalisées

L'éditeur des actions personnalisées permet de définir des portions de code à exécuter pendant la phase d'installation ou de désinstallation. L'exécution est liée à l'un des quatre évènements suivants : **Installer** qui se produit après l'installation des fichiers mais avant la validation. **Valider** qui se produit après que l'installation ait été validée. **Restaurer** qui se produit lorsque l'installation a échoué. **Désinstaller** qui se produit lorsque l'application est désinstallée.

Pour ajouter une action personnalisée, ouvrez le menu contextuel d'un évènement et sélectionnez le menu **Ajouter une action personnalisée...**. Dans la fenêtre qui s'ouvre, sélectionnez l'élément qui devra être exécuté. Il peut s'agir de n'importe quel code exécutable. L'action s'affiche alors dans l'arborescence de l'éditeur :

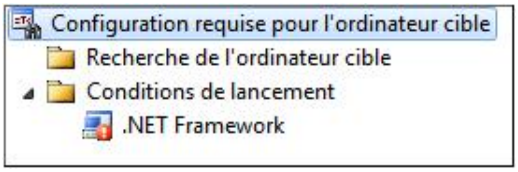


Ouvrez la fenêtre des propriétés pour configurer l'action personnalisée. Les propriétés disponibles sont les suivantes :

- **Arguments** : spécifie les arguments de ligne de commande qui seront passés à l'action personnalisée.
- **Condition** : spécifie une condition qui doit être vraie pour que l'action personnalisée soit exécutée. La condition peut évaluer les valeurs des champs des boîtes de dialogue pour déterminer si l'action doit être effectuée ou non.
- **CustomActionData** : cette propriété permet de passer des données personnalisées à passer à l'action.
- **InstallerClass** : cette valeur booléenne détermine si l'action personnalisée est implémentée dans une classe `Installer`.

f. L'éditeur des conditions de lancement

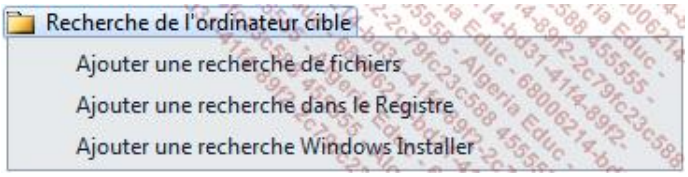
L'éditeur des conditions de lancement permet de spécifier des conditions que les machines cibles devront remplir pour que l'installation puisse se dérouler. L'éditeur est composé de deux éléments : la recherche sur l'ordinateur cible et les conditions de lancement :



Une condition de lancement, **.NET Framework**, est déjà spécifiée par défaut avec la version du Framework correspondant à la version utilisée par l'application.

Vous pouvez ajouter des recherches sur des fichiers, des clés de registre ou Windows Installer. Les conditions pourront alors être basées sur le résultat de ces recherches.

Pour créer une nouvelle recherche, ouvrez le menu contextuel du dossier **Recherche de l'ordinateur cible** et sélectionnez dans le menu le type de recherche à effectuer :



La recherche de fichiers permet de spécifier un fichier qui devra être recherché sur l'ordinateur cible. Les propriétés de ce type de recherche permettent de préciser le nom de fichier, la profondeur de sous-dossiers dans lesquels chercher à partir du dossier de base, lui aussi paramétrable. Vous pouvez affiner la recherche de fichier avec un intervalle de dates, de tailles et de versions.

La recherche dans le registre permet de spécifier une clé de registre qui sera recherchée sur l'ordinateur cible.

La recherche Windows Installer permet de rechercher si un composant est installé sur l'ordinateur cible en précisant l'identifiant de celui-ci.

Chacune des recherches possède une propriété **Property** qui permet de définir le nom de la propriété dans laquelle la valeur booléenne correspondant au résultat de la recherche sera stockée. Ces propriétés peuvent ensuite être utilisées pour les conditions de lancement, dans leur propriété **Condition**. Si la condition est remplie, l'installation continuera, sinon elle échouera.