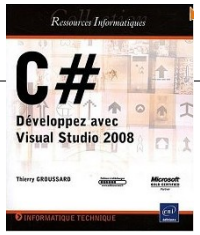
Écriture, compilation et exécution d'une application C#

BTS CIEL	Compétence visée	C08	CODER



D'après:
C# - Développez avec Visual Studio 2008
de Thierry GROUSSARD

1) Écriture du code

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

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

Pour cette première application, nous allons utiliser des outils individuels : le bloc-notes de Windows pour l'écriture du code et le compilateur en ligne de commandes pour Visual C#.
Cette première application sera très simple puisqu'elle affichera simplement le message "Bonjour" dans une fenêtre de commande.
Exemple

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

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

`using System`

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

`class Program`

Dans Visual C#, toute portion de code doit être contenue dans une classe (langage 100 % objet).

`static String message= "Bonjour" ;`

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

`static void Main (String[]args)`

Toutes les instructions autres que des déclarations doivent être placées dans une procédure ou une fonction (en fait une méthode). La majeure partie du code est donc placée entre les caractères { et } délimitant chaque procédure ou fonction. Parmi toutes ces procédures et fonctions, l'une d'entre elles est désignée comme le point d'entrée dans l'application. C'est par l'exécution de cette procédure que démarre l'application. Cette procédure doit s'appeler **Main** et doit être statique. Elle doit être déclarée à l'intérieur d'une classe ou d'une structure. Le type de retour peut être `void` ou `int`. Les paramètres sont optionnels et, s'ils sont utilisés, ils représentent les arguments passés sur la ligne de commande.

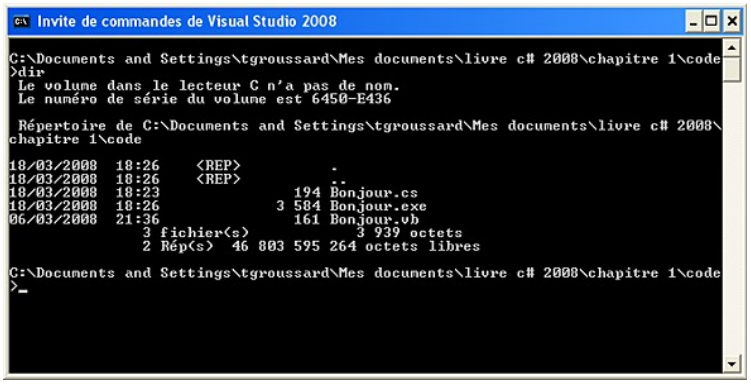
`Console.Writeline("Bonjour") ;`

La classe Console définie dans l'espace de nom `System` fournit un ensemble de méthodes permettant l'affichage d'informations sur la console ou la lecture d'informations depuis la console. La procédure `Writeline` permet l'affichage d'une chaîne de caractères sur la console.
À noter également que Visual C# fait la distinction entre les minuscules et les majuscules dans les instructions. Si vous utilisez l'éditeur de Visual Studio pour rédiger votre code, celui-ci vous guidera pour éviter les erreurs (**IntelliSense**).

2) Compilation du code

Le Framework .NET inclut un compilateur en ligne de commande pour Visual C#. Pour compiler le code source de notre exemple, nous devons ouvrir une fenêtre Invite de commande pour pouvoir lancer le compilateur.

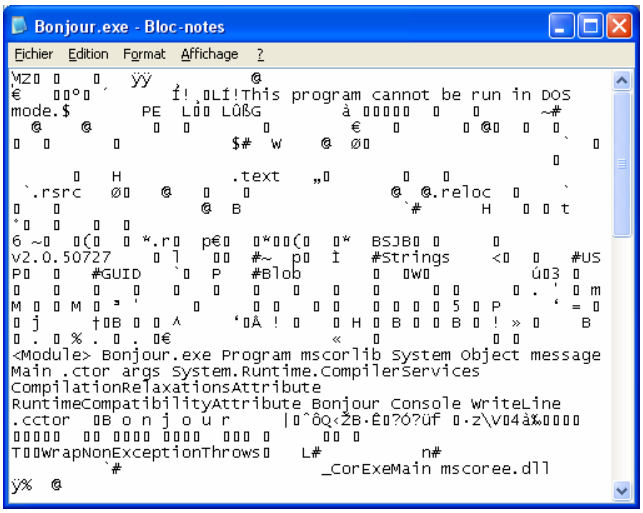
À partir de la fenêtre de commande ouverte, il convient de se placer dans le répertoire dans lequel se trouve le fichier source. **csc.exe** doit être recherché et la compilation est lancée par la commande `csc Bonjour.cs`.
Après un bref instant, le compilateur nous rend la main. Nous pouvons vérifier la présence du fichier exécutable et vérifier son bon fonctionnement.



Notre première application est vraiment très simple. Pour des applications plus évoluées, il sera parfois utile de spécifier certaines options pour le fonctionnement du compilateur. L'ensemble des options disponibles peut être obtenu en lançant la commande `csc / ?` .
Les principales options sont :
`/out:fichier.exe`
Cette option permet de spécifier le nom du fichier résultat de la compilation. Par défaut, c'est le nom du fichier source en cours de compilation qui est utilisé.
`/target :exe`
Cette option demande au compilateur la génération d'un fichier exécutable pour une application en mode console.
`/target :winexe`
Cette option demande au compilateur la génération d'un fichier exécutable d'application Windows.
`/target :library`
Cette option demande au compilateur la génération d'un fichier bibliothèque dll.
`/reference :liste de fichiers`
Cette option indique au compilateur la liste des fichiers référencés dans le code et nécessaires pour la compilation. Les noms des fichiers doivent être séparés par une virgule.

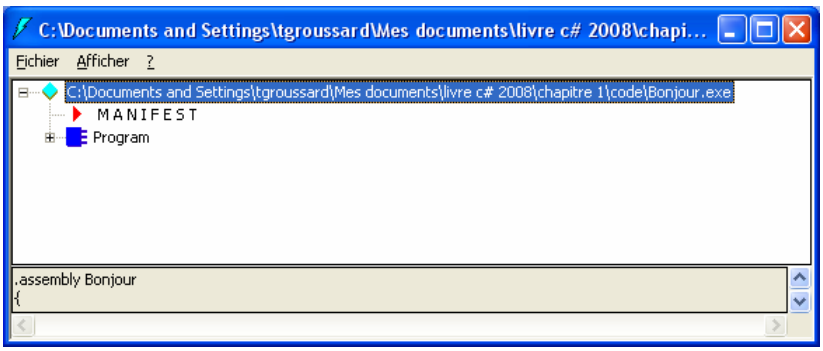
3) Analyse d'un fichier compilé

Maintenant que notre fichier exécutable est créé, essayons de voir ce qu'il contient.
Première solution : l'ouvrir avec le bloc-notes de Windows

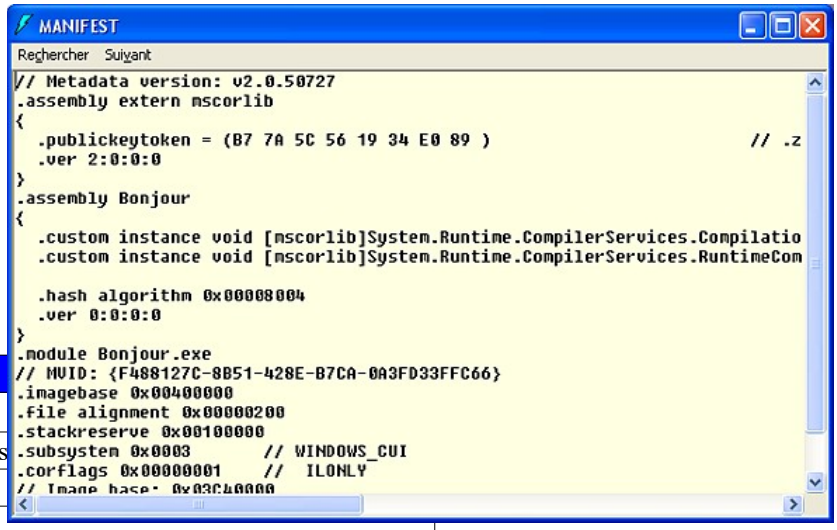


Le résultat n'est pas très parlant, c'est le moins que l'on puisse dire !
Nous avons dit que le compilateur génère du code CIL (commun intermediate language) (ex MSIL). C'est donc ce code que nous visualisons dans le bloc-notes. Pour visualiser le contenu d'un fichier CIL, le framework .NET propose un outil plus adapté : un désassembleur

Cet outil est lancé à partir de la ligne de commande par l'instruction `ildasm`.
Il permet de visualiser, de manière plus claire que le bloc-notes, un fichier généré par le compilateur. Il convient d'indiquer le fichier que l'on souhaite examiner, par le menu Fichier - Ouvrir. Le désassembleur affiche alors son contenu.
Les informations présentes dans le fichier peuvent être séparées en deux catégories : le manifest et le code CIL (ex MSIL). Le manifest contient les métadonnées permettant de décrire le contenu du fichier et les ressources dont il a besoin. On parle dans ce cas de fichier auto descriptif. Cette technique est très intéressante car dès que le Common Language Runtime CLR (la VM) lit le fichier, il dispose de toutes les informations nécessaires pour son exécution.



Le manifest peut être visualisé par un double clic sur son nom.



Nous retrouvons dans ce manifest des informations indiquant que, pour que l'application puisse fonctionner, elle a besoin de l'assembly extern mscorlib.
La deuxième partie correspond réellement au code CIL (ex MSIL). Un ensemble d'icônes est utilisé pour faciliter la visualisation des informations.

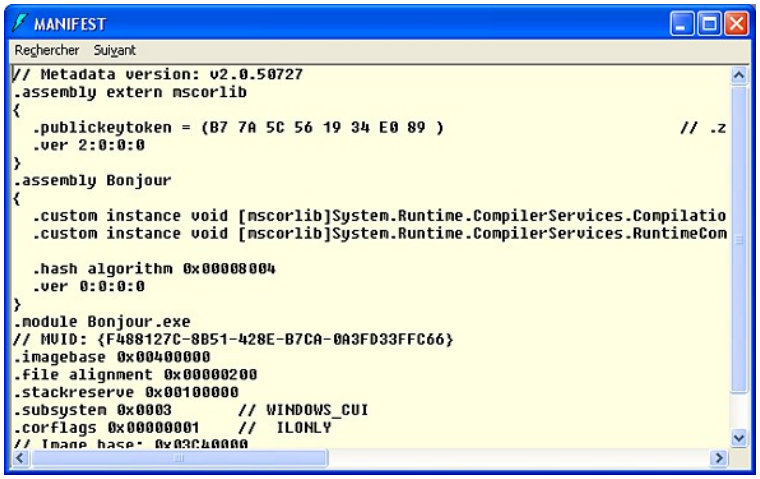
Symbole	Signification
	Plus d'infos
	Espace de noms
	Classe
	Interface
	Classe de valeurs
	Enumération
	Méthode
	Méthode statique
	Champ
	Champ statique
	événement
	Propriété
	élément de manifeste ou d'infos de classe

Comme pour le manifest, un double clic sur un élément permet d'obtenir plus de détails. Ainsi, nous pouvons, par exemple, visualiser la traduction de notre méthode Main.
Dans un exemple de code aussi simple, il est facile de faire la correspondance entre le code Visual C# et sa traduction en code CIL. Pour les personnes enthousiasmées par le code CIL, il existe un assembleur CIL: ilasm. Cet outil attend un fichier texte contenant du code CIL et le transforme en format binaire.

Puisque nous sommes capables de visualiser le code CIL (ex MSIL), nous pouvons vérifier qu'il est bien indépendant du langage source utilisé pour développer l'application.

Voici le code Visual Basic qui réalise la même chose que notre code Visual C#.

```
using System;
    Imports System
Public Module test
    Dim message As String = "Bonjour"
    Public Sub main()
        console.writeline(message)
    End Sub
End Module
```



Après compilation et désassemblage par ildasm, voici ce qu'il nous présente pour la méthode Main.
Il n'y a aucune différence par rapport à la version Visual C# de la méthode Main.
Il est également possible de faire la démarche inverse en transformant un fichier texte contenant du code CIL en fichier binaire correspondant. Cette transformation se fait grâce à l'assembleur `ilasm`.
La seule difficulté est de créer un fichier texte contenant le code CIL car même si la syntaxe est compréhensible, elle n'est pas très intuitive. Une solution peut être de demander à l'outil `ildasm` (le désassembleur) de générer ce fichier texte. Pour cela après, avoir ouvert le fichier exécutable ou la bibliothèque `dll` avec `ildasm`, vous devez utiliser l'option `Dump` du menu Fichier. Vous êtes alors invité à choisir le nom du fichier à générer (extension .il).
Ce fichier peut être ensuite modifié avec un simple éditeur de texte. Remplacez par exemple le contenu de la variable

message avec la chaîne "Hello".

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

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

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

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

4) Exécution du code

Lorsqu'un utilisateur exécute une application managée, le chargeur de code du système d'exploitation charge le Common Language Runtime qui ensuite démarre l'exécution du code managé. Comme le processeur de la machine sur laquelle s'exécute l'application ne peut pas prendre en charge directement le code CIL, Le Common Language Runtime doit le convertir en code natif.
Cette conversion ne concerne pas la totalité du code au chargement de l'application. Il convertit le code au fur et à mesure des besoins. La démarche adoptée est la suivante :

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