Ressourcesinformatiques

+ QUIZ

Version en ligne
OFFERTE !
pendant 1 an

Apprendre la Programmation Orientée Objet avec le langage C#

(avec exercices pratiques et corrigés)

4e édition

Luc GERVAIS

En téléchargement

des exercices
et corrigés

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **RI4CAPOO** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Chapitre 5
Création d'objets

Chapitre 6
Héritage et polymorphisme

Chapitre 7
Communication entre objets

Chapitre 8
Le multithreading

Chapitre 9
P-Invoke

Chapitre 10
Les tests

Chapitre 11

Traçage et instrumentation des applications

Chapitre 12
La réflexion

Chapitre 13
Gestion des données

Chapitre 14
WPF MVVM et le toolkit Microsoft

Chapitre 4
Les types du C#

1. "En C#, tout est typé !"

Le terme générique "**type**" regroupe les classes, les structures, les records, les interfaces, les énumérations et les délégués. Ces types sont décrits dans la CTS (*Common Type System*) pour que des compilateurs de langages différents puissent générer un code exploitable par la CLR (*Common Language Runtime*). Un programme utilise les différents types et un assemblage peut implémenter plusieurs types.

Voici les définitions succinctes des différents types proposés par le C# :

– Le type "Classe" est l'implémentation C# de ce qui a été présenté dans les premiers chapitres. La classe est évidemment le type le plus utilisé dans les applications. Le chapitre Création de classes en définit précisément la syntaxe de déclaration, d'allocation et d'utilisation.

– Le type "Structure" est assez voisin de celui du langage C. Avant la démocratisation de la programmation objet, les structures étaient le moyen le plus commun offert aux développeurs pour composer leurs propres types. Retenons pour l'instant que les structures du C# sont très proches des classes et que, quand elles sont utilisées à bon escient, elles peuvent améliorer les performances d'une application. Nous verrons au chapitre suivant que le .NET encapsule la plupart de ses types "simples" (les entiers, les caractères, etc.) dans des structures. Sachez que les structures n'existent pas en Java.

– Le type "Record" est un objet pouvant être classe ou structure qui s'identifie par son contenu et non par son emplacement en mémoire. Cette distinction subtile sera détaillée plus loin.

– Le type "Interface" est largement utilisé dans le .NET et contribue à la communication entre les classes. Retenons pour l'instant qu'une interface est une classe souvent sans code qui formalise un lot de méthodes obligatoires pour la classe qui l'implémentera. Le chapitre Héritage et polymorphisme traite du sujet.

– Le type "Enumération" permet la définition de listes clés-valeurs et la création des données dont les contenus seront limités à ces clés. Rappelons l'exemple d'un type *Jour* pouvant contenir de *lundi* à *dimanche*. Si, lors de la rédaction du programme, on tente de copier dans un objet de ce type la clé *Mars*, il y aura une erreur de compilation. En C#, ce type apporte un lot de méthodes permettant de gérer cette liste par programmation.

– Le type "Delegate" (Délégué) encapsule la notion de pointeur de fonction du C/C++, origine de bien des soucis, en commençant par lui attribuer un typage fort. En effet, le pointeur de fonction "conventionnel" n'est autre qu'une adresse mémoire sans autre précision sur la signature et l'application s'arrête en erreur quand les paramètres passés ne correspondent pas aux paramètres attendus... C'est pourquoi le type *delegate* du C# va être défini précisément avec la signature de la méthode qui lui sera associée. Ensuite, l'instance de type *delegate*, généralement créée au sein d'une classe amenée à communiquer avec d'autres, gère une liste "d'abonnés" via une syntaxe déconcertante de simplicité. Il suffit en effet d'utiliser l'opérateur += du *delegate* pour s'abonner à la liste de diffusion et -= pour s'en désenregistrer. Les *delegate* sont très largement utilisés dans le C# ; on les retrouvera beaucoup dans les interfaces graphiques pour que les composants puissent notifier l'application de leurs changements d'états.

Durant ce chapitre seront abordées des notions illustrées par des extraits de code. Ces extraits de code utilisent des syntaxes décrites dans les chapitres suivants mais la compréhension des chapitres suivants passe... par celle de ce présent passage ! Vous avez donc à prendre pour argent comptant dans un premier temps les syntaxes des exemples mais nous les approfondirons par la suite.

Remarque

Tous les exemples de ce chapitre sont des projets de la solution Visual Studio TypesDuCSharp.sln figurant dans le répertoire Chap4 du .zip accompagnant cet ouvrage. Ce fichier d'accompagnement est à télécharger sur le site des Éditions ENI www.editions-eni.fr.

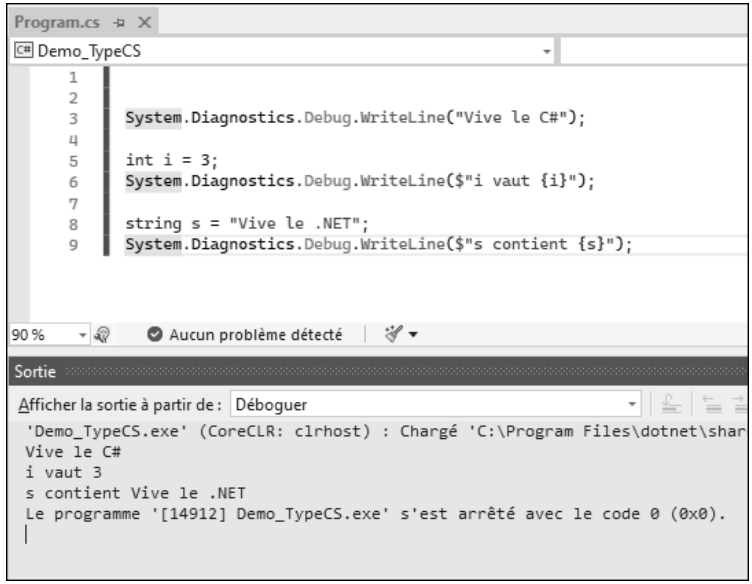
Introduction à System.Diagnostics.Debug

Ces exemples utilisent des classes de tests et également une classe système appelée *System.Diagnostics.Debug*. Cette classe permet, entre autres, d'écrire des messages dans la fenêtre **Sortie** de Visual Studio et également de vérifier des conditions passées en paramètres. Elle sera étudiée avec d'autres classes de même type dans le chapitre Traçage et instrumentation des applications.

Syntaxe d'affichage dans la fenêtre **Sortie** de Visual Studio

```
System.Diagnostics.Debug.WriteLine("le message");
```

Exemple d'utilisations simples et composées



Le signe $ qui précède le contenu de la chaîne permet d'effectuer une "interpolation", à savoir un remplacement de séquence {blabla} par le contenu de la variable blabla. Cette extension très souple et très pratique est arrivée avec le C# 6.

La méthode *System.Diagnostics.Debug.Assert* permet de vérifier qu'une condition est vraie pendant l'exécution de votre code. En utilisant cette méthode vous n'intervenez pas sur le déroulement du programme en tant que tel ; vous vérifiez juste que ce qui est prévu à tel endroit du code est correct. Si la condition est fausse, une boîte de dialogue sera affichée pour vous en informer.

Syntaxe d'utilisation de la méthode System.Diagnostics.Debug.Assert

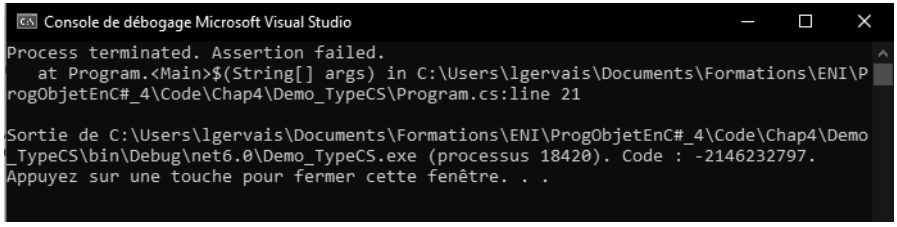
```
System.Diagnostics.Debug.Assert(<condition>);
```

Exemples d'utilisation de la méthode Assert

```
// Vérification de la condition "1 est différent de 2"
System.Diagnostics.Debug.Assert(1 != 2);
// Comme la condition est vraie, le programme
// passe à la ligne suivante


// Pour voir l'effet produit
// lorsqu'une condition n'est pas vérifiée,
// une erreur est "forcée" en ligne suivante
System.Diagnostics.Debug.Assert(1 == 2);
```

À l'exécution de la seconde ligne de l'extrait, le programme affiche une boîte de message et attend que l'utilisateur la referme avant de poursuivre l'exécution du code.

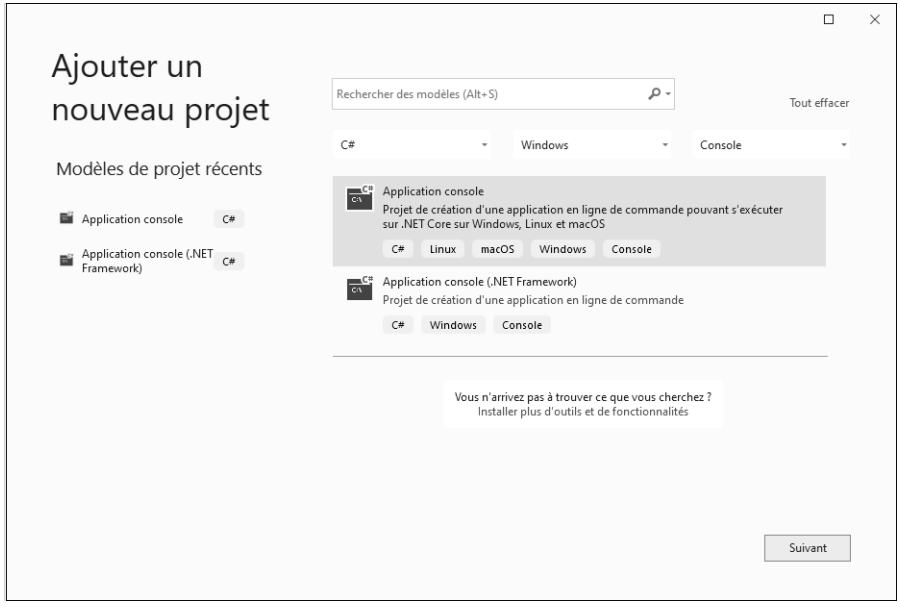


Ainsi, vous pouvez vérifier que ce que vous avez prévu se réalise correctement. On utilise *System.Diagnostics.Debug.Assert* principalement pendant les phases de mise au point pour éventuellement ajouter du code de protection par la suite. Nous verrons d'ailleurs que Visual Studio génère une version "mise au point" (*Debug*) et une version "production" (*Release*). *System.Diagnostics.Debug.Assert* n'a aucun effet sur un code compilé en mode "production".

Introduction à System.Console

Nous l'avons déjà utilisée au chapitre Introduction à .NET 6 et à VS pour afficher le classique *Hello World* à l'écran ; la console va servir de support à plusieurs exemples à suivre. Cet environnement d'exécution très sommaire présente l'avantage de pouvoir simplement afficher des chaînes à l'écran et lire des entrées clavier.

Comme nous l'avons vu, le type **Application console** se choisit à la création du projet.



Voici les principales commandes qui seront utilisées :

Affichage d'une chaîne suivie d'un changement de ligne

```
Console.WriteLine("Message à afficher...");
```

Affichage d'un type primitif sans changement de ligne

```
      int j = 358;
      Console.Write(j);
```

Affichage d'une composition

```
int k = 2;
int l = 3;
Console.WriteLine($"k contient {k} et l contient {l}");
```

Lecture d'une chaîne de caractères saisie au clavier
et terminée par la touche [Entrée]

```
string saisie = Console.ReadLine();
```

Ces présentations étant faites, nous pouvons passer à la suite...

2. "Tout le monde hérite de System.Object"

Le type *System.Object* est la base directe ou indirecte de tous les types du .NET, ceux existants et ceux que vous allez créer (la notion d'héritage a déjà été un peu abordée dans les premiers chapitres). L'héritage d'*Object* étant implicite, sa déclaration est inutile. Tous les types héritent de ses méthodes et peuvent même en substituer certaines.

C'est ce que fait *System.ValueType* qui, dans la hiérarchie des types du .NET, devient la base de la famille "Valeurs" en adaptant les méthodes de *System.Object*.

2.1 Les types Valeurs

La famille "Valeurs" se divise en plusieurs parties :

– les énumérations
– les structures
– les records (s'ils sont instanciés en type Valeur)

Les structures sont elles-mêmes sous-divisées en :

– types numériques :
  – les types intégraux :

Type	Taille
sbyte	Entier signé sur 8 bits
byte	Entier non signé sur 8 bits
char	Caractère UNICODE 16 bits
short	Entier signé sur 16 bits
ushort	Entier non signé sur 16 bits
int	Entier signé sur 32 bits
uint	Entier non signé sur 32 bits
long	Entier signé sur 64 bits
ulong	Entier non signé sur 64 bits

  – les types à virgule flottante :

Type	Précision
float	7 chiffres
double	15-16 chiffres

  – le type décimal (adapté aux calculs financiers) :

Type	Précision
decimal	28-29 chiffres