



Ressourcesinformatiques

 + QUIZ

Version en ligne

OFFERTE !

pendant 1 an

Apprendre la Programmation Orientée Objet avec le langage C#

(avec exercices pratiques et corrigés)

4^e édition

Luc GERVAIS

En téléchargement



des exercices
et corrigés



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.



Avant-propos

Chapitre 1 Introduction à la POO

- 1. Histoire de la POO 15
- 2. Historique du C# 18

Chapitre 2 La conception orientée objet

- 1. Approche procédurale
et décomposition fonctionnelle 19
- 2. La transition vers l'approche objet. 20
- 3. Les caractéristiques de la POO. 21
 - 3.1 L'objet, la classe et la référence 21
 - 3.1.1 L'objet 21
 - 3.1.2 La classe. 22
 - 3.1.3 La référence 23
 - 3.2 L'encapsulation 24
 - 3.3 L'héritage. 24
 - 3.4 Le polymorphisme 26
 - 3.5 L'abstraction 27
- 4. Le développement objet 28
 - 4.1 Cahier des charges du logiciel 28
 - 4.2 Présentation du cycle en V. 28

2 _____ Apprendre la POO

avec le langage C#

4.3	Modélisation et représentation UML	32
4.3.1	Diagrammes de cas d'utilisation	35
4.3.2	Diagrammes de classes	36
4.3.3	Énumérations	42
4.3.4	Diagrammes de séquences	43
4.4	Rédaction du code et des tests unitaires	45
5.	Exercices corrigés	46
5.1	Hiérarchie de classes	46
5.2	Relations entre objets	48
5.3	Agrégation d'objets	48
5.4	Diagramme de cas d'utilisation	50
5.5	Diagramme de séquences	51

Chapitre 3

Introduction à .NET 6 et à VS

1.	Introduction	53
2.	Environnement d'exécution	54
3.	Le choix des langages	55
4.	Utiliser plusieurs langages	55
5.	Une librairie très complète	55
6.	Un outil de développement complet	57

Chapitre 4

Les types du C#

1.	"En C#, tout est typé !"	67
2.	"Tout le monde hérite de System.Object"	72
2.1	Les types Valeurs	73
2.2	Les types Références	76
2.3	Boxing/unboxing	78

- 2.4 Utilisation des méthodes de System.Object 79
 - 2.4.1 Equals 80
 - 2.4.2 GetHashCode 84
 - 2.4.3 ToString 86
 - 2.4.4 Finalize 87
 - 2.4.5 Object.GetType et les opérateurs typeof et is 88
 - 2.4.6 object.ReferenceEquals. 89
 - 2.4.7 Object.MemberwiseClone 90
- 2.5 Le type System.String et son alias string 93
- 3. Exercice corrigé 97
 - 3.1 Énoncé 97
 - 3.2 Correction 97

Chapitre 5
Création d'objets

- 1. Introduction 101
- 2. Les espaces de noms 102
- 3. Déclaration d'une classe 110
 - 3.1 Accessibilité des membres 112
 - 3.2 Attributs 112
 - 3.2.1 Attributs constants 113
 - 3.2.2 Attributs en lecture seule 114
 - 3.3 Propriétés. 116
 - 3.4 Constructeur. 124
 - 3.4.1 Étapes de la construction d'un objet 124
 - 3.4.2 Surcharge de constructeurs 126
 - 3.4.3 Constructeurs avec valeurs de paramètres par défaut . . 126
 - 3.4.4 Chaînage de constructeurs. 127
 - 3.4.5 Les constructeurs de type static 128
 - 3.4.6 Les constructeurs de type private 129
 - 3.4.7 Les initialiseurs d'objets 131

4 _____ Apprendre la POO

avec le langage C#

3.5	Destructeur	132
3.6	Autre utilisation de using	134
3.7	Le mot-clé this et ses vertus	135
3.8	Méthodes	138
3.8.1	Déclaration	139
3.8.2	Passage par valeur et passage par référence	142
3.9	Mécanisme des exceptions	159
3.9.1	Présentation	159
3.9.2	Principe de fonctionnement des exceptions	160
3.9.3	Prise en charge de plusieurs exceptions	169
3.9.4	try ... catch ... finally et using	170
3.10	Surcharge des méthodes	172
3.11	Exercice	174
3.11.1	Énoncé	174
3.11.2	Conseils	175
3.11.3	Correction	175
4.	Les interfaces	178
4.1	Introduction	178
4.2	Le contrat	178
4.3	Déclaration d'une interface	179
4.4	Implémentation	181
4.5	Visual Studio et les interfaces	183
4.6	Représentation UML d'une interface	186
4.7	Interfaces et polymorphisme	187
4.8	Exercice	188
4.8.1	Énoncé	188
4.8.2	Conseils	188
4.8.3	Correction	191
4.9	Les interfaces du .NET	194

5. Association, composition et agrégation	196
5.1 Les tableaux.	204
5.2 Les collections	212
5.2.1 List<> et LinkedList<>	213
5.2.2 Queue<T> et Stack<T>	216
5.2.3 Dictionary<TKey, TValue>	217
5.2.4 Les énumérateurs	217
5.2.5 La magie du yield	219
5.3 Exercice	220
5.3.1 Énoncé.	220
5.3.2 Correction	222
6. Les classes imbriquées.	223
7. Les structures	225
7.1 Déclaration d'une structure	226
7.2 Instanciation d'une structure.	228
8. Les classes partielles	230
9. Les méthodes partielles.	231
10. Les indexeurs.	233
11. Surcharge d'opérateurs	237
12. Fonctions locales.	240
13. Les objets "gourmands" en références faibles	241
14. Les objets "dynamics"	243
15. Les "Tuple" et "ValueType"	244
16. Les records.	245
16.1 Introduction	245
16.2 Déclaration complète	246
16.3 Déclaration simplifiée.	247
16.4 Comparaison de records.	249
16.5 Déconstruction d'un record.	251
16.6 Mutation d'un record.	251

6 --- Apprendre la POO

avec le langage C#

Chapitre 6

Héritage et polymorphisme

1. Comprendre l'héritage	253
2. Codage de la classe de base et de son héritière	254
2.1 Interdire l'héritage	254
2.2 Définir les membres héritables	255
2.3 Codage de l'héritage	255
2.4 Exploitation d'une classe héritière	256
3. Communication entre classe de base et classe héritière	257
3.1 Les constructeurs	257
3.2 Accès aux membres de base depuis l'héritier	260
3.3 Masquage ou substitution de membres hérités	262
3.3.1 Codage du masquage	264
3.3.2 Codage de la substitution	266
4. Exercice	267
4.1 Énoncé	267
4.2 Corrigé	268
5. Les classes abstraites	269
6. Les méthodes d'extension	271
7. Le polymorphisme	274
7.1 Comprendre le polymorphisme	274
7.2 Exploitation du polymorphisme	275
7.3 Les opérateurs is, as et ()	275

Chapitre 7

Communication entre objets

1. L'événementiel : être à l'écoute	279
2. Le pattern Observateur	280

- 3. La solution C# : delegate et event. 284
 - 3.1 Utilisation du delegate dans le design pattern Observateur . . 287
 - 3.2 Utilisation d'un event. 290
 - 3.3 Comment accompagner l'event de données 293
 - 3.4 Les génériques en renfort pour encore simplifier 295
 - 3.5 Les expressions lambda 296
 - 3.6 Exemple d'utilisation d'event. 301
- 4. Appels synchrones, appels asynchrones 309
- 5. Exercice 311
 - 5.1 Énoncé 311
 - 5.2 Conseils pour la réalisation 312
 - 5.3 Correction 312
- 6. Des messages entre les classes 316

Chapitre 8
Le multithreading

- 1. Introduction 317
- 2. Comprendre le multithreading 318
- 3. Multithreading et .NET 321
- 4. Implémentation en C# 322
 - 4.1 Utilisation d'un BackgroundWorker 322
 - 4.1.1 Communication du thread principal
vers le thread secondaire 324
 - 4.1.2 Abandon du thread secondaire
depuis le thread principal. 325
 - 4.1.3 Communication du thread secondaire
vers le thread principal. 326
 - 4.1.4 Communication en fin de traitement
du thread secondaire. 326
 - 4.1.5 Exemple de code 327
 - 4.2 Utilisation du pool de threads créé par .NET 329

4.3	Gestion "manuelle" avec Thread/ParameterizedThreadStart . . .	331
5.	Synchronisation entre threads	336
5.1	Nécessité de la synchronisation	336
5.2	Le mot-clé lock	338
5.3	La classe Monitor	339
5.4	La classe Mutex	340
5.5	La classe Semaphore	341
6.	Communication entre threads	342
6.1	Join	342
6.2	Les synchronization events	343
6.3	Communication entre threads secondaires et IHM	350
6.4	Exercice	353
6.4.1	Énoncé	353
6.4.2	Correction	353
7.	La programmation asynchrone	357
7.1	Les « Task »	357
7.2	async et await	359
7.3	Le mot-clé async	360
7.4	Contenu d'une méthode async	360
7.5	Preuve à l'appui	360
7.6	Retours possibles d'une méthode async	362

Chapitre 9

P-Invoke

1.	Introduction	365
1.1	Rappel sur les DLL non managées	366
1.2	P-Invoke et son Marshal	366
2.	Le cas simple	367
2.1	Déclaration et appel	368
2.2	Réglage de Visual Studio pour la mise au point	370
3.	Appel avec paramètres et retour de fonction	371

- 4. Traitement avec des chaînes de caractères 373
 - 4.1 Encodage des caractères 373
 - 4.2 Encodage des chaînes 374
 - 4.3 Transmission des chaînes..... 375
- 5. Échange de tableaux 378
 - 5.1 Du C# au C/C++ 378
 - 5.2 Du C# au C/C++ puis retour au C# 380
- 6. Partage de structures 381
 - 6.1 Déclaration des structures 381
 - 6.2 Utilisation des structures..... 383
- 7. Les directives [In] et [Out]..... 388
- 8. Réalisation d'un wrapper 392
 - 8.1 Une région "NativeMethods"..... 393
 - 8.2 Stockage des informations de la DLL native..... 394
 - 8.3 Instanciation de DLL native..... 395
 - 8.4 Méthodes d'utilisation de la DLL managée
depuis le wrapper 397
 - 8.5 Utilisation du wrapper..... 398
- 9. Exercice 399
 - 9.1 Énoncé..... 399
 - 9.2 Correction..... 400

Chapitre 10
Les tests

- 1. Introduction 403
- 2. Environnement d'exécution des tests unitaires 405
- 3. Le projet de tests unitaires 408
- 4. La classe de tests 409
- 5. Contenu d'une méthode de test 410
- 6. Traitements de préparation et de nettoyage 413

7. DynamicData et source de données	417
8. Automatisation des tests à la compilation	422
9. Automatisation des tests en dehors de Visual Studio	423
10. CodedUI	425
11. Exercice	426
11.1 Énoncé	426
11.2 Correction	426
12. Simulation par stub ou par shim	428

Chapitre 11

Traçage et instrumentation des applications

1. Présentation	431
2. Des objets de mise au point	432
2.1 System.Diagnostics.Debug	432
2.2 System.Diagnostics.Trace	435
2.3 System.Diagnostics.TraceSource	436
3. Principe de fonctionnement des écouteurs	437
4. Comportement dynamique	439
5. Mesurer le temps passé	443
6. Exercice	446
6.1 Énoncé	446
6.2 Correction	446

Chapitre 12
La réflexion

- 1. Introduction 449
- 2. Mais pour quoi faire ? 450
- 3. Introspection d'une classe C# 452
 - 3.1 Introspection "manuelle" 455
 - 3.2 Introspection "logicielle" 458
 - 3.2.1 Découverte et instanciation 458
 - 3.2.2 Découverte et utilisation des propriétés 461
 - 3.2.3 Découverte et utilisation des méthodes 463
 - 3.3 Exercice 466
 - 3.3.1 Énoncé 466
 - 3.3.2 Quelques conseils 466
 - 3.3.3 Correction 466
- 4. Chargement dynamique d'un objet implémentant une interface. 469
 - 4.1 Création d'une interface "plug-in" 470
 - 4.2 Écriture d'un plug-in 471
 - 4.3 L'application supportant les plug-ins 473
 - 4.4 Exercice 475
 - 4.4.1 Énoncé 475
 - 4.4.2 Correction 475
- 5. Décompilation et obfuscation 477
- 6. Conclusion 483

Chapitre 13 Gestion des données

1. Introduction	485
2. LINQ	486
2.1 Qu'est-ce que c'est ?	486
2.2 Les deux syntaxes LINQ	487
2.2.1 La syntaxe "développeur SQL"	487
2.2.2 La syntaxe "développeur C#"	489
2.3 Requêtes et filtres	489
2.4 Quelques calculs	492
2.5 Regroupement des résultats	494
2.6 Les jointures	496
2.7 Exercice	498
2.7.1 Énoncé	498
2.7.2 Solution	499
3. Persistance des données en XML	506
3.1 Rappels sur le XML	507
3.2 XML et .NET	509
3.2.1 Sérialisation/désérialisation d'un modèle de données ..	509
3.2.2 Les décorations de sérialisation XML	509
3.2.3 XmlSerializer : écrire et lire	513
3.3 XSD.EXE, un outil de conversion	516
3.4 Exercice	518
3.4.1 Énoncé	518
3.4.2 Correction	518
3.5 LINQ to XML	520
3.5.1 Lecture	521
3.5.2 Écriture	522
3.5.3 Interrogations	524
3.6 Exercice	524
3.6.1 Énoncé	524
3.6.2 Correction	524

- 4. Persistance dans des bases de données avec ADO.NET 526
 - 4.1 Présentation 526
 - 4.2 Les termes utilisés 526
 - 4.3 Les modules ADO.NET 527
 - 4.4 Notre environnement d'apprentissage 528
 - 4.5 ADO en mode connecté 532
 - 4.5.1 Les fournisseurs de données en .NET 532
 - 4.5.2 Se connecter avec DbConnection 533
 - 4.5.3 Envoyer des requêtes avec DbCommand 537
 - 4.5.4 Lire des enregistrements avec DbDataReader 543
 - 4.6 Exercice 546
 - 4.6.1 Énoncé 546
 - 4.6.2 Correction 546
 - 4.7 ADO en mode déconnecté 548
 - 4.7.1 La classe DataSet 548
 - 4.7.2 Le DataSet typé 551
 - 4.7.3 Persistance du DataSet en XML 559
 - 4.7.4 LINQ to DataSet 561
 - 4.7.5 Intégrité référentielle 562
 - 4.8 DbAdapter : jonction des deux modes 569
 - 4.8.1 Lecture de la source 570
 - 4.8.2 Mise à jour de la source 573
- 5. Entity Framework 580
 - 5.1 Présentation de l'Entity Data Model 581
 - 5.2 Création d'un EDM depuis une base de données 582
 - 5.3 DbContext 588
 - 5.4 LINQ to Entities 594
 - 5.5 Mise à jour de la source 596
 - 5.6 Création d'un EDM depuis un modèle 600
 - 5.7 Exercice 612
 - 5.7.1 Présentation du binding 612
 - 5.7.2 Énoncé 613
 - 5.7.3 Correction 613

14 _____ Apprendre la POO

avec le langage C#

6. Conclusion	614
-------------------------	-----

Chapitre 14 WPF MVVM et le toolkit Microsoft

1. Présentation	615
2. Historique des API	616
3. C# et XAML Développeur et Graphiste	617
4. Balises et Attributs pour Objets et Propriétés	617
5. Utilisation basique de WPF	622
6. Utilisation des layout	626
7. Récupération des informations SANS le binding	633
8. Introduction au binding	639
8.1 Le DataContext	639
8.2 L'interface INotifyPropertyChanged	642
8.3 Les convertisseurs	645
8.4 Exercice	648
8.5 Binding de commandes	649
9. Le modèle de conception MVVM	656
9.1 Objectifs	656
9.2 Les dépendances	657
9.3 Mise en application	658
10. Présentation de MVVM Toolkit	669
10.1 La classe Observable	669
10.2 La classe ObservableValidator	671
10.3 Messenger	672
10.4 L'injection de dépendances	675
Index	681

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 "Énumération" 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 type 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.

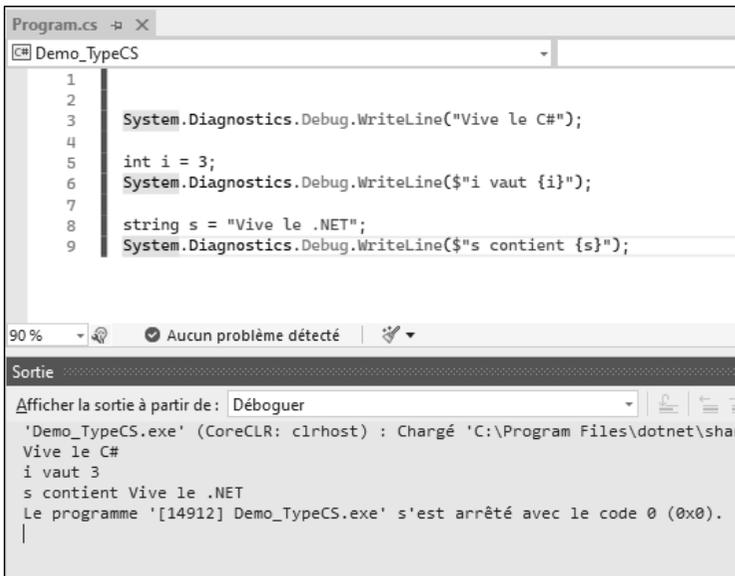
Introduction à `System.Diagnostics.Debug`

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

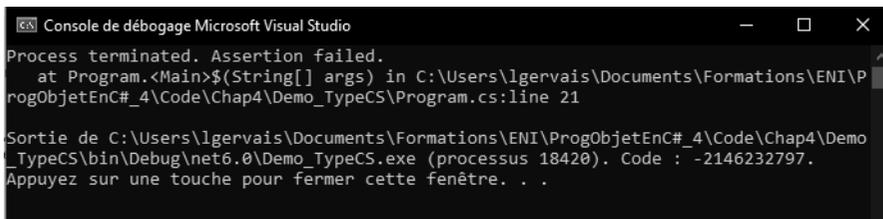
```
System.Diagnostics.Debug.Assert(<condition>);
```

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.



```
Microsoft Visual Studio
Process terminated. Assertion failed.
   at Program.<Main>$(String[] args) in C:\Users\lgervais\Documents\Formations\ENI\ProgObjetEnC#_4\Code\Chap4\Demo_TypeCS\Program.cs:line 21

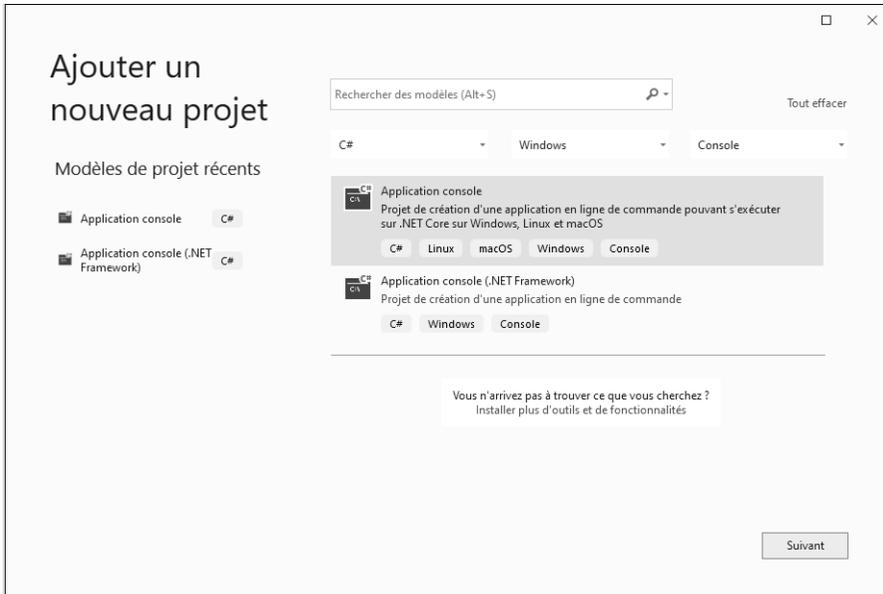
Sortie de C:\Users\lgervais\Documents\Formations\ENI\ProgObjetEnC#_4\Code\Chap4\Demo_TypeCS\bin\Debug\net6.0\Demo_TypeCS.exe (processus 18420). Code : -2146232797.
Appuyez sur une touche pour fermer cette fenêtre. . .
```

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