



Ressourcesinformatiques

Apprendre la Programmation Orientée Objet avec le langage C#

2^e édition

Luc GERVAIS

Téléchargement
www.editions-eni.fr



Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI de l'ouvrage **RI2CAPOO** 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 11
- 2. Historique du C# 14

Chapitre 2 La conception orientée objet

- 1. Approche procédurale et décomposition fonctionnelle. 15
- 2. La transition vers l'approche objet. 16
- 3. Les caractéristiques de la POO. 17
 - 3.1 L'objet, la classe et la référence 17
 - 3.1.1 L'objet 17
 - 3.1.2 La classe. 18
 - 3.1.3 La référence 19
 - 3.2 L'encapsulation 20
 - 3.3 L'héritage. 20
 - 3.4 Le polymorphisme 22
 - 3.5 L'abstraction 23
- 4. Le développement objet 24
 - 4.1 Cahier des charges du logiciel 24

2 _____ Apprendre la POO

avec le langage C#

4.2	Modélisation et représentation UML	24
4.2.1	Diagrammes de cas d'utilisation	27
4.2.2	Diagrammes de classes	28
4.2.3	Énumérations	34
4.2.4	Diagrammes de séquences	35
4.3	Rédaction du code et tests unitaires	37

Chapitre 3

Introduction au framework .NET et à VS

1.	Introduction	39
2.	Environnement d'exécution	39
3.	Le choix des langages	40
4.	Utiliser plusieurs langages	40
5.	Une librairie très complète	41
6.	Des installations simplifiées	42
7.	Un outil de développement complet	44

Chapitre 4

Les types du C#

1.	"En C#, tout est typé !"	51
2.	"Tout le monde hérite de System.Object"	56
2.1	Les types Valeurs	56
2.2	Les types Référence	60
2.3	Boxing - Unboxing	61
2.4	Utilisation des méthodes de System.Object	62
2.4.1	Equals	63
2.4.2	GetHashCode	66
2.4.3	ToString	68
2.4.4	Finalize	69
2.4.5	Object.GetType et les opérateurs typeof et is	70

- 2.4.6 Object.ReferenceEquals 71
- 2.4.7 Object.MemberwiseClone 72
- 2.5 Le type System.String et son alias string 75
- 3. Exercice corrigé 79
 - 3.1 Énoncé 79
 - 3.2 Correction 79

Chapitre 5
Création de classes

- 1. Introduction 83
- 2. Les espaces de noms 83
- 3. Déclaration d'une classe 90
 - 3.1 Accessibilité des membres 91
 - 3.2 Attributs 92
 - 3.2.1 Attributs constants 93
 - 3.2.2 Attributs en lecture seule 94
 - 3.3 Propriétés 95
 - 3.4 Constructeur 103
 - 3.4.1 Étapes de la construction d'un objet 103
 - 3.4.2 Surcharge de constructeurs 105
 - 3.4.3 Constructeurs avec valeurs de paramètres par défaut . 105
 - 3.4.4 Chaînage de constructeurs 106
 - 3.4.5 Les constructeurs de type static 107
 - 3.4.6 Les constructeurs de type private 108
 - 3.4.7 Les initialiseurs d'objets 110
 - 3.5 Destructeur 111
 - 3.6 Autre utilisation de using 113
 - 3.7 Le mot-clé this 114
 - 3.8 Méthodes 117
 - 3.8.1 Déclaration 117
 - 3.8.2 Passage par valeur et passage par référence 121

4 --- Apprendre la POO

avec le langage C#

3.9	Mécanisme des exceptions	136
3.9.1	Présentation	136
3.9.2	Principe de fonctionnement des exceptions	137
3.9.3	Prise en charge de plusieurs exceptions	147
3.9.4	try ... catch ... finally et using	148
3.10	Surcharge des méthodes	152
3.11	Exercice	154
3.11.1	Énoncé	154
3.11.2	Conseils	155
3.11.3	Correction	155
4.	Les interfaces	158
4.1	Introduction	158
4.2	Le contrat	159
4.3	Déclaration d'une interface	160
4.4	Implémentation	161
4.5	Visual Studio et les interfaces	163
4.6	Représentation UML d'une interface	165
4.7	Interfaces et polymorphisme	165
4.8	Exercice	166
4.8.1	Énoncé	166
4.8.2	Conseils	167
4.8.3	Correction	169
4.9	Les interfaces du .NET	173
5.	Association, composition et agrégation	175
5.1	Les tableaux	182
5.2	Les collections	189
5.2.1	List<> et LinkedList<>	190
5.2.2	Queue<T> et Stack<T>	193
5.2.3	Dictionary<TKey, TValue>	193
5.2.4	Les énumérateurs	194
5.2.5	La magie du yield	195

- 5.3 Exercice 197
 - 5.3.1 Énoncé..... 197
 - 5.3.2 Correction..... 198
- 6. Les classes imbriquées..... 200
- 7. Les structures 203
 - 7.1 Déclaration d'une structure..... 203
 - 7.2 Instanciation d'une structure..... 205
- 8. Les classes partielles 208
- 9. Les méthodes partielles..... 209
- 10. Les indexeurs..... 210
- 11. Surcharge d'opérateurs 214

Chapitre 6
Héritage et polymorphisme

- 1. Comprendre l'héritage 219
- 2. Codage de la classe de base et de son héritière 220
 - 2.1 Interdire l'héritage 220
 - 2.2 Définir les membres héritables 221
 - 2.3 Codage de l'héritage 221
 - 2.4 Exploitation d'une classe héritée 222
- 3. Communication entre classe de base
 et classe héritière 223
 - 3.1 Les constructeurs 223
 - 3.2 Accès aux membres de base depuis l'héritier..... 227
 - 3.3 Masquage ou substitution de membres hérités 229
 - 3.3.1 Codage du masquage 231
 - 3.3.2 Codage de la substitution 232
- 4. Exercice 234
 - 4.1 Énoncé..... 234
 - 4.2 Corrigé..... 235

6 --- Apprendre la POO

avec le langage C#

5. Les classes abstraites	237
6. Les méthodes d'extension	238
7. Le polymorphisme	241
7.1 Comprendre le polymorphisme	241
7.2 Exploitation du polymorphisme	242
7.3 Les opérateurs is et as	242

Chapitre 7

Communication entre objets

1. L'événementiel : être à l'écoute	243
2. Le pattern Observateur	244
3. La solution C# : delegate et event	248
3.1 Utilisation du delegate dans le design pattern Observateur	251
3.2 Utilisation d'un event	254
3.3 Comment accompagner l'event de données	257
3.4 Les génériques en renfort pour encore simplifier	258
3.5 Les expressions lambda	259
3.6 Exemple d'utilisation d'event	265
4. Appels synchrones, appels asynchrones	272
4.1 Approche 1	275
4.2 Approche 2	276
4.3 Approche 3	276
4.4 Approche 3 avec une expression lambda	279
5. Exercice	281
5.1 Énoncé	281
5.2 Conseils pour la réalisation	282
5.3 Correction	282

Chapitre 8
Le multithreading

- 1. Introduction 287
- 2. Comprendre le multithreading 287
- 3. Multithreading et .NET 290
- 4. Implémentation en C# 292
 - 4.1 Utilisation d'un BackgroundWorker 292
 - 4.1.1 Communication du thread principal
vers le thread secondaire 295
 - 4.1.2 Abandon depuis le thread primaire 296
 - 4.1.3 Communication du thread secondaire
vers le thread principal 296
 - 4.1.4 Communication en fin de traitement
du thread secondaire 296
 - 4.1.5 Exemple de code 297
 - 4.2 Utilisation du pool de threads créé par .NET 299
 - 4.3 Gestion « manuelle »
avec Thread/ParameterizedThreadStart302
- 5. Synchronisation entre threads 307
 - 5.1 Nécessité de la synchronisation 307
 - 5.2 La décoration Synchronization 309
 - 5.3 Le mot-clé lock 311
 - 5.4 La classe Monitor 312
 - 5.5 La classe Mutex 313
 - 5.6 La classe Semaphore 315
- 6. Communication entre threads 315
 - 6.1 Join 315
 - 6.2 Les synchronization events 317
 - 6.3 Communication entre threads secondaires et IHM 324
 - 6.4 Exercice 328
 - 6.4.1 Énoncé 328
 - 6.4.2 Correction 328

8 --- Apprendre la POO

avec le langage C#

7. La programmation asynchrone	333
7.1 Le mot-clé async	333
7.2 Contenu d'une méthode async	333
7.3 Preuve à l'appui.	334
7.4 Retours possibles d'une méthode async	335

Chapitre 9 P-Invoke

1. Introduction	339
1.1 Rappel sur les DLL non managées	340
1.2 P-Invoke et son Marshal.	340
2. Le cas simple	341
2.1 Déclaration et appel	342
2.2 Réglage de Visual Studio pour la mise au point	345
3. Appel avec paramètres et retour de fonction	345
4. Traitement avec des chaînes de caractères	348
4.1 Encodage des caractères	348
4.2 Encodage des chaînes	349
4.3 Transmission des chaînes.	349
5. Échange de tableaux	352
5.1 Du C# au C/C++	352
5.2 Du C# au C/C++ puis retour au C#	354
6. Partage de structures	356
6.1 Déclaration des structures	356
6.2 Utilisation des structures.	359
7. Les directives [In] et [Out].	362
8. Réalisation d'un "wrapper"	366
8.1 Une région "NativeMethods".	367
8.2 Stockage des informations de la DLL native.	368
8.3 Instanciation de DLL native	369
8.4 Méthodes d'utilisation de la DLL managée	

depuis le wrapper 372

8.5 Utilisation du wrapper 373

9. Exercice 374

9.1 Énoncé 374

9.2 Correction 375

Chapitre 10
Les tests

1. Introduction 377

2. Environnement d'exécution des tests unitaires 379

3. Le projet de tests unitaires 382

4. La classe de tests 383

5. Contenu d'une méthode de test 384

6. Traitements de préparation et de nettoyage 387

7. TestContext et source de données 390

8. Automatisation des tests à la compilation 394

9. Automatisation des tests en dehors de Visual Studio 395

10. CodedUI 396

11. Exercice 397

11.1 Énoncé 397

11.2 Correction 398

Index 401

Chapitre 2

La conception orientée objet

1. Approche procédurale et décomposition fonctionnelle

Avant d'énoncer les bases de la programmation objet nous allons revoir l'approche procédurale à l'aide d'un exemple concret d'organisation de code.

La programmation procédurale est un paradigme de programmation considérant les différents acteurs d'un système comme des objets pratiquement passifs qu'une procédure centrale utilisera pour une fonction donnée.

Prenons l'exemple de la distribution d'eau courante dans nos habitations et essayons d'en émuler le principe dans une application très simple. L'analyse procédurale (tout comme l'analyse objet d'ailleurs) met en évidence une liste d'objets qui sont :

- le robinet de l'évier ;
- le réservoir du château d'eau ;
- un capteur de niveau d'eau avec contacteur dans le réservoir ;
- la pompe d'alimentation puisant l'eau dans la rivière.

Le code du programme "procédural" consisterait à créer un ensemble de variables représentant les paramètres de chaque composant puis d'écrire une boucle de traitement de gestion centrale testant les valeurs lues et agissant en fonction du résultat des tests. On notera qu'il y a d'un côté les variables et de l'autre, les actions.

2. La transition vers l'approche objet

La programmation objet est un paradigme de programmation considérant les différents acteurs d'un système comme des objets actifs et en relation. L'approche objet est souvent très voisine de la réalité.

Dans notre exemple, l'utilisateur ouvre le robinet ; le robinet relâche la pression et l'eau s'écoule du réservoir à l'évier ; le capteur/flotteur du réservoir arrive à un niveau qui déclenche la pompe ; l'utilisateur referme le robinet ; alimenté par la pompe, le réservoir du château d'eau se remplit et le capteur/flotteur atteint un niveau qui arrête la pompe.

Dans cette approche, vous constatez que les objets interagissent ; il n'existe pas de traitement central définissant dynamiquement le fonctionnement des objets. Il y a eu, en amont, une analyse fonctionnelle qui a conduit à la création des différents objets, à leurs montages et à leurs mises en relations.

Le code du programme "objet" va suivre cette réalité en proposant autant d'objets que décrit précédemment mais en définissant entre ces objets des méthodes d'échange adéquates qui conduiront au fonctionnement escompté.

■ Remarque

Les concepts objets sont très proches de la réalité...

3. Les caractéristiques de la POO

3.1 L'objet, la classe et la référence

3.1.1 L'objet

L'objet est l'élément de base de la POO. L'objet est l'union :

- d'une liste de variables d'états,
- d'une liste de comportements,
- d'une identification.

Les variables d'états changent durant la vie de l'objet. Prenons le cas d'un lecteur de musiques numériques. À l'achat de l'appareil, les états de cet objet pourraient être :

- mémoire libre = **100%**
- taux de charge de la batterie = **faible**
- aspect extérieur = **neuf**

Au fur et à mesure de son utilisation, ses états vont être modifiés. Rapidement la mémoire libre va chuter, le taux de charge de la batterie variera et l'aspect extérieur va changer en fonction du soin apporté par l'utilisateur.

Les comportements de l'objet définissent ce que peut faire cet objet : Jouer la musique - Aller à la piste suivante - Aller à la piste précédente - Monter le son, etc. Une partie des comportements de l'objet est accessible depuis l'extérieur de cet objet : bouton Play, bouton Stop... et une autre partie est uniquement interne : lecture de la carte mémoire, décodage de la musique à partir du fichier. On parle "d'encapsulation" pour définir une limite entre les comportements accessibles de l'extérieur et les comportements internes.

L'identification d'un objet est une information, détachée de la liste des états, permettant de différencier l'objet de ses congénères (c'est-à-dire d'autres objets qui ont le même type que lui). L'identification peut être un numéro de référence ou une chaîne de caractères unique construite à la création de l'objet ; elle peut également être une adresse mémoire. En réalité, sa forme dépend totalement de la problématique associée.

L'objet POO peut être attaché à une entité bien réelle, comme pour notre lecteur numérique, mais il peut être également attaché à une entité totalement virtuelle comme un compte client, l'entrée d'un répertoire téléphonique... La finalité de l'objet informatique est de gérer l'entité physique ou de l'émuler.

Même si ce n'est pas dans sa nature de base, l'objet peut être rendu persistant c'est-à-dire que ses états peuvent être enregistrés sur un support mémorisant l'information de façon intemporelle. À partir de cet enregistrement, l'objet pourra être recréé quand le système le souhaitera.

3.1.2 La classe

La classe est le "moule" à partir duquel l'objet va être créé en mémoire. La classe contient les états et les comportements communs d'un même type et les valeurs de ces états seront contenues dans ses objets.

Les comptes courants d'une même banque contiennent tous les mêmes paramètres (coordonnées du détenteur, solde...) et ont tous les mêmes fonctions (créditer, débiter...). Ces définitions doivent être contenues dans une classe et, à chaque fois qu'un client ouvre un nouveau compte, cette classe servira de modèle à la création de l'objet compte.

Le présentoir de lecteurs de musiques numériques propose un même modèle en différentes couleurs, avec des tailles mémoire modulables, etc. Chaque appareil du présentoir est un objet qui a été fabriqué à partir des informations d'une seule classe. À la réalisation, les attributs de l'appareil ont été choisis en fonction de critères esthétiques et commerciaux.

Une classe peut contenir beaucoup d'attributs. Ils peuvent être de type primitif – des entiers, des caractères... – mais également de type plus complexe. En effet, une classe peut contenir une ou plusieurs classes d'autres types. On parle alors de composition ou encore de "couplage fort". La destruction de la classe principale entraîne, évidemment, la destruction des classes qu'elle contient. Par exemple, si une classe *hôtel* contient une liste de *chambres*, la destruction de l'*hôtel* entraîne la destruction des *chambres*.

Une classe peut faire "référence" à une autre classe ; dans ce cas, le couplage est dit "faible" et les objets peuvent vivre indépendamment. Par exemple, votre PC est relié à votre imprimante. Si votre PC rend l'âme, votre imprimante fonctionnera certainement avec le futur PC ! On parle d'association.

En plus de ses attributs, la classe contient également une série de "comportements", c'est-à-dire une série de méthodes avec signatures et code attaché. Ces méthodes sont directement "copiées" dans les objets et utilisées telles quelles.

On déclare la classe et son contenu dans un même fichier source à l'aide d'une syntaxe que l'on étudiera en détail. Les développeurs C++ apprécieront le fait qu'il n'existe plus, d'un côté, une partie définitions et, de l'autre, une partie implémentations. En effet, en C#, le fichier programme (d'extension .CS) contient les définitions de tous les états avec éventuellement une valeur "de départ" et les définitions et implémentations de tous les comportements. On verra qu'il existe en C# une solution permettant de définir une classe dans plusieurs fichiers afin que des intervenants puissent travailler en parallèle sans jamais effacer le travail de l'autre.

3.1.3 La référence

Les objets sont construits à partir de la classe, par un processus appelé l'instanciation, et donc, tout objet est une instance d'une classe. Chaque instance commence à un emplacement mémoire unique. Cet emplacement mémoire connu sous le nom de *pointeur* par les développeurs C et C++ devient une *référence* pour les développeurs C# et Java.

Quand le développeur a besoin d'un objet pendant un traitement, il doit :

- déclarer et nommer une variable du type de la classe à utiliser ;
- instancier l'objet et enregistrer sa référence dans cette variable.

Une fois cette instanciation réalisée, le programme accédera aux propriétés et aux méthodes de l'objet par la variable contenant sa référence. Chaque instance est unique. Par contre, plusieurs variables peuvent "pointer" sur une même instance. C'est d'ailleurs quand plus aucune variable ne pointe sur une instance donnée que le ramasse-miettes enregistre cette instance comme devant être détruite.

3.2 L'encapsulation

L'encapsulation consiste à créer une sorte de boîte noire contenant en interne un mécanisme protégé et en externe un ensemble de commandes qui vont permettre de le manipuler. Ce jeu de commandes est fait de telle sorte qu'il sera impossible d'altérer le mécanisme protégé en cas de mauvaise utilisation. La boîte noire sera si opaque qu'il sera impossible à l'utilisateur d'intervenir en direct sur le mécanisme.

Vous l'aurez compris, la boîte noire n'est autre qu'un objet avec des méthodes publiques de "haut niveau" contrôlant avec rigueur les paramètres passés avant de les utiliser dans un traitement. En respectant le principe de l'encapsulation, vous ferez en sorte que jamais l'utilisateur de l'objet ne puisse accéder "en direct" à vos données. Grâce à ce contrôle, votre objet sera correctement utilisé, donc plus fiable, et sa mise au point sera plus facile. Dans le monde Java, les méthodes permettant d'accéder en lecture aux données sont des accesseurs et celles permettant d'accéder en écriture sont des mutateurs. Nous verrons que C# propose une solution très élégante, les propriétés, qui permet de garder le côté pratique de l'accès direct aux données de l'objet tout en respectant les principes de l'encapsulation.

3.3 L'héritage

Une autre notion importante de la POO concerne l'héritage. Pour l'expliquer, imaginons que nous devons construire un système de gestion des différents types de collaborateurs d'une société. Une analyse rapide met en évidence une liste de propriétés communes à tous les postes. En effet, que le collaborateur soit ouvrier, cadre, intérimaire ou encore directeur, il a toujours un nom, un prénom et un numéro de sécurité sociale... On appelle cela la généralisation ; elle consiste à factoriser les éléments communs d'un ensemble de classes dans une classe plus générale appelée superclasse en Java et plutôt "classe de base" en C# et C++. La classe qui hérite de la superclasse est appelée sous-classe ou classe héritière.