



Ressourcesinformatiques

 + QUIZ

Version en ligne

OFFERTE !

pendant 1 an

C# 10

et Visual Studio Code

Les fondamentaux du langage

En téléchargement



code source

Christophe MOMMER





Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI de l'ouvrage **RI10CSHAVSC** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

1. Introduction 1

Chapitre 1

Introduction

1. Qu'est-ce que C# ? 9
1.1 Que peut-on réaliser avec C# ? 10
1.2 Le langage est-il stable et pérenne ? 12
2. Préparer son environnement 13
2.1 Installation et configuration de Visual Studio Code 14
2.2 Installer les outils de compilation 16
3. Comment fonctionne le C# ? 19

Chapitre 2

Premier programme

1. Créer sa première application C# 23
2. Comprendre et écrire du code C# 27
2.1 Notions de variable et constante 29
2.1.1 Types numériques 31
2.1.2 Types textuels 33
2.1.3 Valeur booléenne 35
2.1.4 Opérateurs 35

2 _____ C# 10 et Visual Studio Code

Les fondamentaux du langage

2.2	Les autres types	38
2.2.1	Stockage des dates	38
2.2.2	Les intervalles de temps	40
3.	Analyser la structure d'un projet C#	41
3.1	La notion de blocs	42
3.2	Signification des blocs de code	44
3.2.1	Le bloc d'espace de noms	44
3.2.2	La définition d'une classe	47
3.2.3	La définition d'une méthode	48
3.3	Déclaration "top-level"	48
4.	Exécuter un programme C#	49
4.1	Lancer le programme avec Visual Studio Code	49
4.2	Lancer depuis la ligne de commande	51
5.	Exercice	54
5.1	Énoncé	54
5.2	Corrigé	54

Chapitre 3

Programmation orientée objet

1.	Principes de la programmation orientée objet	57
1.1	Qu'est-ce qu'une classe ?	57
1.1.1	Les classes dans Visual Studio Code	59
1.1.2	L'héritage	60
1.1.3	L'encapsulation	61
1.2	Que peut-on déclarer dans une classe ?	62
1.2.1	Les méthodes	62
1.2.2	Déclarer une donnée	65
1.3	Instancier une classe	69
1.3.1	Le constructeur	69
1.3.2	L'instanciation avec le mot-clé new	71
1.4	Le polymorphisme	73

- 2. Concepts avancés 75
 - 2.1 L'héritage avancé. 75
 - 2.1.1 Méthodes virtuelles 75
 - 2.1.2 Classe abstraite 76
 - 2.1.3 Interface 78
 - 2.1.4 Implémentation par défaut dans une interface 80
 - 2.1.5 Masquage 81
 - 2.1.6 Interdire l'héritage 82
 - 2.2 Les différents types d'objets. 83
 - 2.2.1 Les types références 83
 - 2.2.2 Les types valeurs 84
 - 2.2.3 Les types nullable 87
 - 2.2.4 Les types références nullable 88
 - 2.2.5 Les énumérations 90
 - 2.2.6 Les records 91
 - 2.3 Les modificateurs de classe. 93
 - 2.3.1 La notion de static 94
 - 2.3.2 La notion de classe partielle 95
- 3. Exercice 96
 - 3.1 Énoncé. 96
 - 3.2 Corrigé. 97

Chapitre 4
Algorithmique

- 1. Bases de l'algorithmique 101
 - 1.1 La logique conditionnelle 101
 - 1.1.1 Test simple : le if/else. 102
 - 1.1.2 Multiples tests avec l'instruction switch 108
 - 1.1.3 Pattern matching 110
 - 1.1.4 Exercice - énoncé. 115
 - 1.1.5 Exercice - corrigé. 115

4 _____ C# 10 et Visual Studio Code

Les fondamentaux du langage

1.2	Les collections	116
1.2.1	L'interface IEnumerable	116
1.2.2	Les tableaux	117
1.2.3	La liste	119
1.2.4	Les dictionnaires	122
1.2.5	Les collections algorithmiques	124
1.3	Les boucles	126
1.3.1	Les généralités sur les boucles	126
1.3.2	La boucle for	127
1.3.3	La boucle while	129
1.3.4	La boucle do while	129
1.3.5	La boucle foreach	130
1.3.6	Le mot-clé yield	130
1.3.7	Exercice - énoncé	131
1.3.8	Exercice - corrigé	132
2.	Gestion des erreurs	135
2.1	Concept d'une exception	135
2.2	Renvoyer une exception	136
2.3	Gérer une exception	139
2.3.1	Blocs try, catch et finally	139
2.3.2	Filtre sur bloc catch	141
2.4	Exceptions et performances	143

Chapitre 5

LINQ

1.	Fonctionnement de base	145
2.	Variables anonymes	148
3.	Principes des opérateurs LINQ	148
3.1	Opérateurs de production	152
3.2	Opérateurs de sélection	163
3.3	Opérateurs de génération	169

- 4. Expression de requête LINQ 169
 - 4.1 Le mot-clé into 170
 - 4.2 Le mot-clé let 172
- 5. Exercice 173
 - 5.1 Énoncé..... 173
 - 5.2 Corrigé..... 174

Chapitre 6
Sérialisation

- 1. Sérialisation en C# 175
- 2. Sérialisation binaire 176
 - 2.1 Utilisation des attributs 177
 - 2.2 Utilisation de l'interface ISerializable 181
- 3. Sérialisation XML 183
 - 3.1 XmlSerializer 183
 - 3.2 XmlDocument, XElement et XAttribute 186
- 4. Sérialisation JSON 190
 - 4.1 Utf8JsonReader et Utf8JsonWriter 191
 - 4.2 JsonDocument 194
 - 4.3 JsonSerializer 195
- 5. Exercice 199
 - 5.1 Énoncé..... 199
 - 5.2 Corrigé..... 201

6 _____ C# 10 et Visual Studio Code

Les fondamentaux du langage

Chapitre 7

Concepts avancés

1. Asynchronisme	203
1.1 Fonctionnement de base	203
1.2 Thread et asynchronisme	205
1.3 Asynchronisme en C#	206
1.4 Les mots-clés async et await	207
1.5 Flux asynchrones	210
2. Algorithmique avancée	212
2.1 Programmation événementielle	212
2.1.1 Les delegates	212
2.1.2 Les events	214
2.2 Les types génériques	217
2.2.1 Utilisation standard	217
2.2.2 Contraintes sur type générique	219
2.3 Gestion de la mémoire	220
2.3.1 Le destructeur	221
2.3.2 IDisposable et IAsyncDisposable	222
2.4 Paramètres de méthodes avancés	223
2.4.1 Paramètre optionnel	223
2.4.2 Mots-clés de paramètres	224
2.4.3 Nommage de paramètres	227
2.4.4 Paramètres variables	228
2.5 Extension du fonctionnement d'un type	228
2.5.1 Méthodes d'extension	229
2.5.2 Définition des opérateurs	230
2.6 Tuples et déconstruction	233
2.6.1 Les tuples en C# 7	233
2.6.2 Déconstruction de type	235
2.7 Fonction locale	237

Chapitre 8
Créer des applications

- 1. Application web 241
 - 1.1 Applications web graphiques..... 241
 - 1.1.1 ASP.NET MVC..... 242
 - 1.1.2 ASP.NET Razor Pages 247
 - 1.1.3 Blazor..... 252
 - 1.2 API..... 256
- 2. Application de bureau 262
 - 2.1 WinForms 263
 - 2.2 Windows Presentation Foundation (WPF)..... 269
 - 2.3 Universal Windows Platform (UWP) 273
- 3. Application mobile 278
 - 3.1 Installation 279
 - 3.1.1 Installation à partir de la ligne de commande 279
 - 3.1.2 Installation avec Visual Studio 2022..... 283
 - 3.2 Code..... 288
- 4. Conclusion 290

Chapitre 9
Référence

- 1. Introduction 291
- 2. Mots-clés de type 291
- 3. Mots-clés de programmation orientée objet..... 293
- 4. Mots-clés algorithmiques..... 297

- Index 303

Chapitre 4

Algorithmique

1. Bases de l'algorithmique

Jusqu'à présent, nous nous sommes contentés de développer des applications n'incluant aucune "logique" : elles se limitaient à afficher des données. Il n'y avait aucune notion de condition, de répétition ou même de logique de code. En effet, le code d'une application est souvent complexe et les embranchements sont multiples en fonction de diverses conditions. Dans ce chapitre, nous allons découvrir la logique algorithmique, qui vous permettra de créer du code plus proche de ce que l'on peut retrouver dans les applications répondant à des problématiques plus complexes.

1.1 La logique conditionnelle

Indéniablement, il s'agit ici d'une brique que vous allez utiliser de façon systématique. Une condition implique l'exécution ou non d'une partie du code en fonction de l'évaluation d'un test logique.

1.1.1 Test simple : le if/else

La logique conditionnelle se traduit en pseudocode de la façon suivante :

```
SI une condition ALORS
    Je fais quelque chose
SINON
    Je fais autre chose
```

En C#, les mots-clés pour réaliser une instruction conditionnelle sont `if` et `else` :

```
if(condition)
{
    ....
}
else
{
    ....
}
```

La condition testée par une instruction `if` doit renvoyer un booléen. Ce dernier peut être stocké dans une variable mais il est également possible que l'instruction `if` évalue directement la condition, sans variable intermédiaire.

Si on reprend l'exemple de la fin du chapitre précédent, on pourrait améliorer notre classe `Voiture` pour rajouter un booléen qui indique si l'instance de la voiture est fonctionnelle. Si la valeur est égale à "oui", il est inutile de réparer la voiture. Cependant, si la voiture n'est pas fonctionnelle, il faut la réparer :

```
public class Voiture
{
    public bool Fonctionnelle { get; set; }
    ...
}
public class Garage
{
    public void Repare(Voiture voiture)
    {
        if(voiture.Fonctionnelle)
        {
            Console.WriteLine("La voiture n'a pas besoin d'être
réparée car elle est fonctionnelle");
        }
    }
}
```

```
        else
        {
            Console.WriteLine("Réparation de la voiture");
            voiture.Fonctionnelle = true;
        }
    }
}
```

Comme on le voit dans le code ci-dessus, l'instruction `if` se base sur la valeur booléenne stockée dans la propriété `Fonctionnelle` de la classe `voiture` pour évaluer si oui ou non la réparation est nécessaire. Ici, le test a été fait de telle sorte que l'on vérifie si la condition est vraie, et dans le cas inverse, on effectue la réparation. On peut très bien inverser la condition initiale, en comparant le booléen à la valeur `false`. De ce fait, on peut même se passer du `else`, qui n'apporte pas réellement de plus-value :

```
public void Repare(Voiture voiture)
{
    if(voiture.Fonctionnelle == false)
    {
        Console.WriteLine("Réparation de la voiture");
        voiture.Fonctionnelle = true;
    }
}
```

À noter également qu'il est possible d'inverser la valeur d'un booléen en mettant un point d'exclamation en préfixe. Ainsi, `!true` est égal à `false`, et `!false` est égal à `true`. Même si cela peut sembler compliqué de prime abord, vous verrez que c'est une façon d'écrire qui deviendra rapidement automatique à l'utilisation. Si l'on reprend l'exemple précédent, le code qui utilise l'inversion de valeur avec le point d'exclamation serait le suivant :

```
public void Repare(Voiture voiture)
{
    if(!voiture.Fonctionnelle)
    {
        Console.WriteLine("Réparation de la voiture");
        voiture.Fonctionnelle = true;
    }
}
```

Même si à première vue l'instruction `else` est utilisée pour définir le cas inverse de celui du `if` principal, elle peut également servir de base pour une autre instruction `if` à suivre afin de faire une instruction ayant pour sémantique "sinon si". Il suffit dans ce cas d'ajouter une condition `if` après le `else`. Par exemple :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
        Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

À noter que la structure du code conditionnel est très flexible : on peut avoir uniquement une seule instruction `if`, une instruction `if` et son `else` associé, ou un enchaînement de `if` et `else if` (avec ou sans `else final`). La seule impossibilité : avoir une instruction `else` seule, car cette dernière indique forcément l'inverse d'une condition donnée.

■ Remarque

Pour que ces embranchements soient possibles, il faut bien sûr qu'il y ait des conditions pouvant donner plusieurs résultats. À ce titre, il n'est pas utile de faire un `if`, `else if`, `else` avec un simple booléen, car ce dernier ne pouvant avoir que deux états, un `if` avec un `else` est suffisant.

Une instruction `if` peut être "compressée" en l'exprimant sous une forme réduite appelée ternaire. Généralement, on utilise cette approche afin d'écrire en ligne un test pour éviter une lourdeur syntaxique, et ce afin d'affecter le contenu d'une variable. La syntaxe est la suivante : on définit en première partie le test à évaluer, séparant d'un point d'interrogation le test des résultats. Ensuite, les cas vrai et faux sont tous deux séparés par un deux-points. La syntaxe est la suivante :

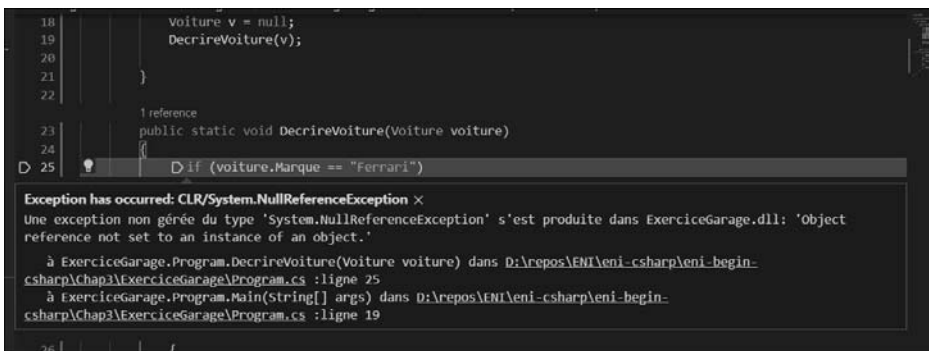
```
test ? cas si vrai : cas si faux
```

Par exemple :

```
string voiture = voiture.Marque == "Ferrari" ? "Voiture chère" :  
"Voiture peu chère";
```

Il est possible d'enchaîner les ternaires avec l'usage des parenthèses (en refaisant une autre ternaire dans un cas ou l'autre) mais il est recommandé d'agir avec parcimonie afin de conserver une lisibilité de code optimale.

Il est souvent intéressant de tester si un objet a été affecté avant d'accéder à ses données ou à ses méthodes. En l'absence de ce test, cela peut provoquer une erreur à l'exécution (appelée exception, que nous détaillerons dans ce chapitre à la section La gestion des erreurs). Si on reprend le code précédent, étant donné que `Voiture` est une classe, elle peut donc valoir la valeur `null`. La fonction `DecrireVoiture` tenterait dès lors d'accéder à une variable qui n'a pas de valeur, provoquant une erreur à l'exécution :



```
18     voiture v = null;  
19     DecrireVoiture(v);  
20  
21 }  
22  
23     1 reference  
24     public static void DecrireVoiture(Voiture voiture)  
25     {  
26         if (voiture.Marque == "Ferrari")
```

Exception has occurred: CLR/System.NullReferenceException ×
Une exception non gérée du type 'System.NullReferenceException' s'est produite dans ExerciceGarage.dll: 'Object reference not set to an instance of an object.'
à ExerciceGarage.Program.DecrireVoiture(Voiture voiture) dans D:\repos\ENI\eni-csharp\eni_begin-csharp\Chap3\ExerciceGarage\Program.cs :ligne 25
à ExerciceGarage.Program.Main(String[] args) dans D:\repos\ENI\eni-csharp\eni_begin-csharp\Chap3\ExerciceGarage\Program.cs :ligne 19

Erreur à l'exécution

Afin d'effectuer un quelconque test sur une donnée d'une classe ou de faire un appel de méthode, il est recommandé de tester si la valeur est bien différente de null. Cela peut se faire de façon "classique" ou grâce au nouvel apport du mot-clé not de C# 9 (comme cela est décrit dans la section à venir Pattern matching) :

```
public void DecrireVoiture(Voiture voiture)
{
    if (voiture != null) // avant C# 9
    {
        ...
    }
    if (voiture is not null) // depuis C# 9
    {
        ...
    }
}
```

Pour éviter ce genre de problèmes, un opérateur de navigation sécurisé a été ajouté en C# 6. Ce dernier permet de n'accéder à une méthode ou de lire une donnée que si la variable n'est pas null. On utilise le point d'interrogation juste après la variable, avant l'appel, et cela permet de se passer de tester la nullité :

```
public void DecrireVoiture(Voiture voiture)
{
    if(voiture?.Marque == "Ferrari")
    {
        Console.WriteLine("Voiture chère");
    }
    else if(voiture?.Marque == "Peugeot")
    {
        Console.WriteLine("Voiture standard");
    }
    else
    {
        Console.WriteLine("Marque de voiture non reconnue");
    }
}
```

Le fonctionnement de cet opérateur est le suivant :

- Si la variable n'est pas `null`, on accède à la propriété ou à la méthode concernée normalement.
- Si la variable est `null` :
 - S'il s'agit d'un appel d'une méthode, et que la méthode ne renvoie rien, elle ne sera pas invoquée ;
 - S'il s'agit d'un appel d'une méthode et que la méthode renvoie une valeur, ou qu'il s'agit d'un appel à une propriété, il faudrait tester si la valeur est différente de `null`. S'il s'agit d'un type référence (d'une classe, comme un `string`), alors il faudrait tester si c'est `null` ou pas, afin de voir si l'appel a été fait. S'il s'agit d'un type valeur, alors le type sera encadré d'un nullable. Par exemple, si le type de retour était un `int`, on obtiendrait un `int?` lors de l'appel, qui serait égal à `null` si la variable était à `null`, ou qui aurait la valeur le cas contraire.

```
public class TestClass
{
    public int Valeur { get; set; }
    public string ValeurString { get; set; }
    public void Methode() { }
    public int MethodeInt()
    {
        return 42;
    }
    public string MethodeString()
    {
        return "valeur";
    }
}

TestClass c = null;
int? valeur = c?.Valeur;
string valeurStr = c?.ValeurString;
c?.Methode();
int? retour = c?.MethodeInt();
string retourStr = c?.MethodeString();
```