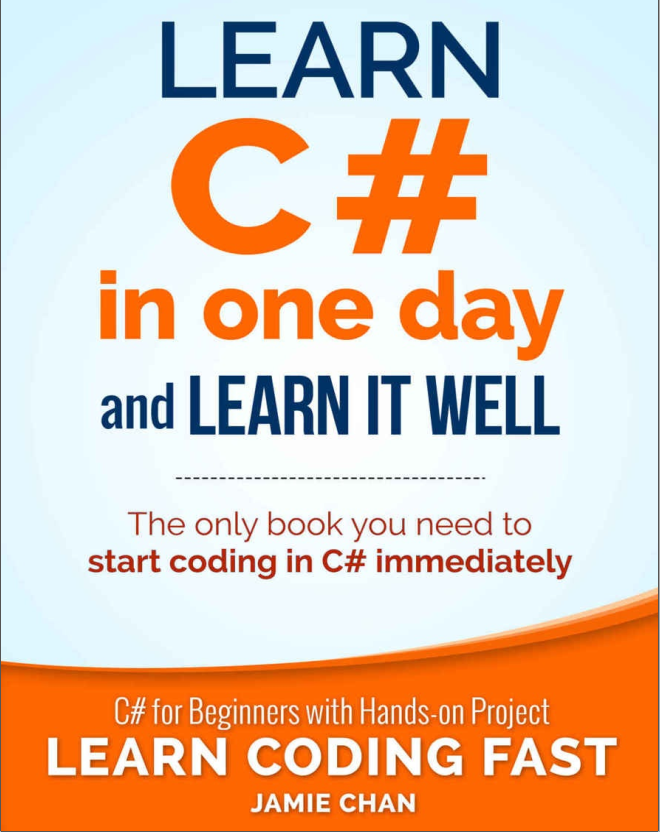
****

Table des matières

[**Chapitre 1: Introduction à C#** 3](#_Toc52663782)

[**Chapitre 2: Se préparer pour C#** 4](#_Toc52663783)

[**Installation de Visual Studio Community** 4](#_Toc52663784)

[**Votre premier programme C#** 4](#_Toc52663785)

[**Structure de base d'un programme C#** 6](#_Toc52663786)

[La méthode Main () 8](#_Toc52663787)

[**Commentaires** 9](#_Toc52663788)

[**Chapitre 3: Le monde des variables et des opérateurs** 10](#_Toc52663789)

[**Que sont les variables?** 10](#_Toc52663790)

[**Types de données en C#** 10](#_Toc52663791)

[**Int (entier)** 10](#_Toc52663792)

[Byte (octet) 10](#_Toc52663793)

[**float (flottant)** 11](#_Toc52663794)

[Double 11](#_Toc52663795)

[Decimal (décimal) 11](#_Toc52663796)

[**Char (caractère)** 11](#_Toc52663797)

[Bool (booléen) 11](#_Toc52663798)

[**Nommer une variable** 12](#_Toc52663799)

[**Initialisation d'une variable** 12](#_Toc52663800)

[Le signe d'affectation 14](#_Toc52663801)

[**Opérateurs de base** 15](#_Toc52663802)

[**Plus d'opérateurs d'affectation** 16](#_Toc52663803)

[**Type de conversion en C#** 18](#_Toc52663804)

[**Chapitre 4 : Tableaux, chaînes et listes** 19](#_Toc52663805)

[**Tableau** 19](#_Toc52663806)

[**Propriétés et méthodes des tableaux** 20](#_Toc52663807)

[**Propriétés et méthodes des chaînes de caractères** 23](#_Toc52663808)

[**Les listes** 26](#_Toc52663809)

[**Propriétés et méthodes de la liste** 27](#_Toc52663810)

[**Type de valeur vs. Type de référence (Type vs. Reference Type)** 29](#_Toc52663811)

[**Chapitre 5 : Rendre notre programme interactif** 30](#_Toc52663812)

[**Afficher des messages aux utilisateurs** 30](#_Toc52663813)

[**Séquences d'évasion (Escape Sequences)** 35](#_Toc52663814)

[**Accepter les données de l'utilisateur** 37](#_Toc52663815)

[**Tout mettre ensemble** 38](#_Toc52663816)

[**Chapitre 6 : Faire des choix et prendre des décisions** 40](#_Toc52663817)

[**Déclarations de condition** 40](#_Toc52663818)

[**Contrôler les déclarations de flux ou de condition** 42](#_Toc52663819)

[**La déclaration If (Si)** 42](#_Toc52663820)

[**Inline If (Si en ligne appelé aussi ternaire)** 45](#_Toc52663821)

[**Switch (déclaration de commutation)** 46](#_Toc52663822)

[**La boucle Pour (For Loop)** 49](#_Toc52663823)

[**La boucle foreach (foreach loop)** 51](#_Toc52663824)

[**La boucle Tant que (While loop)** 51](#_Toc52663825)

[**Faire Tant que (Do while)** 52](#_Toc52663826)

[**Instructions de saut (Jump Statements)** 53](#_Toc52663827)

[**Traitement des exceptions(try-catch-finally)** 56](#_Toc52663828)

[**Erreurs spécifiques** 58](#_Toc52663829)

[**Chapitre 7 : Programmation Orientée Objet - Partie 1** 60](#_Toc52663830)

[**Qu'est-ce que la programmation orientée objet ?** 60](#_Toc52663831)

[**Écrire notre propre classe** 61](#_Toc52663832)

[**Concepts de méthodes avancées** 80](#_Toc52663833)

[**Paramètres de Type Référence vs Type de Valeur de Passage** 84](#_Toc52663834)

[**Chapitre 8 : Programmation axée sur les objectifs - Partie 2** 87](#_Toc52663835)

[**Héritage** 87](#_Toc52663836)

[**Écriture de la classe parent** 87](#_Toc52663837)

[**Ecriture de la classe enfant** 88](#_Toc52663838)

[**La méthode Main()** 94](#_Toc52663839)

[**Polymorphisme** 95](#_Toc52663840)

[**GetType() et typeof()** 98](#_Toc52663841)

[**Classes et méthodes d'abstraction** 99](#_Toc52663842)

[**Interfaces** 102](#_Toc52663843)

[**Révision des modificateurs d'accès** 104](#_Toc52663844)

[**Chapitre 9 : Enum et Struct** 107](#_Toc52663845)

[**Enum** 107](#_Toc52663846)

[**Struct** 109](#_Toc52663847)

[**Chapitre 10 : LINQ** 112](#_Toc52663848)

[**Chapitre 11 : Traitement des dossiers** 116](#_Toc52663849)

[**Lecture d'un fichier texte** 116](#_Toc52663850)

[**Option 1 : try...catch** 119](#_Toc52663851)

[**Option 2 : File.Exists()** 119](#_Toc52663852)

[**Ecrire dans un fichier texte** 120](#_Toc52663853)

[**Projet - Un logiciel de paie simple** 123](#_Toc52663854)

[**Vue d'ensemble** 123](#_Toc52663855)

[**La class Staff** 124](#_Toc52663856)

[**La classe Manager (Manager : Staff)** 127](#_Toc52663857)

[**La classe Admin (Admin : Staff)** 129](#_Toc52663858)

[**La classe FileReader** 131](#_Toc52663859)

[**La classe PaySlip (bulletins de paie)** 133](#_Toc52663860)

[**La classe Program** 139](#_Toc52663861)

# **Chapitre 1: Introduction à C#**

Bienvenue dans la programmation C# et merci beaucoup d'avoir choisi ce livre!

Que vous soyez un programmeur chevronné ou un novice complet, ce livre est écrit pour vous aider à apprendre rapidement la programmation C#. Les sujets sont soigneusement sélectionnés pour vous donner une large exposition à C# sans vous submerger par une surcharge d'informations.

À la fin du livre, vous ne devriez avoir aucun problème à écrire vos propres programmes C#. En fait, nous coderons ensemble un logiciel de paie simple dans le cadre du projet à la fin du livre. Prêt à commencer?

Tout d'abord, répondons à quelques questions:

**Qu'est-ce que C#?**

C#, prononcé C Sharp, est un langage de programmation orienté objet développé par Microsoft au début des années 2000, dirigé par Anders Hejlsberg. Il fait partie du framework .Net et se veut un simple langage de programmation à usage général qui peut être utilisé pour développer différents types d'applications, y compris des applications console, Windows, Web et mobiles.

Comme tous les langages de programmation modernes, le code C# ressemble à la langue anglaise que les ordinateurs sont incapables de comprendre. Par conséquent, le code C# doit être converti en langage machine à l'aide de ce que l'on appelle un compilateur (reportez-vous à la note de bas de page). Le compilateur que nous utiliserons dans ce livre est le logiciel gratuit Visual Studio Community 2015 fourni par Microsoft.

**Pourquoi apprendre C#?**

C# a une syntaxe et des fonctionnalités qui ressemblent à d'autres langages de programmation comme Java et C++. En tant que tel, si vous avez une expérience de programmation préalable, vous trouverez l'apprentissage de C# un jeu d'enfant. Même si vous êtes totalement novice en programmation, C# est conçu pour être facile à apprendre (contrairement au C ou C++) et est un excellent premier langage à apprendre.

De plus, C# fait partie du framework .Net. Ce framework comprend une grande bibliothèque de code pré-écrit que les programmeurs peuvent utiliser sans avoir à tout écrire à partir de zéro. Cela permet aux programmeurs de développer rapidement leurs applications en C#, faisant de C# le langage idéal avec lequel travailler si vous avez un calendrier serré.

Enfin, C# est un langage de programmation orientée objet (POO). La programmation orientée objet est une approche de la programmation qui divise un problème de programmation en objets qui interagissent les uns avec les autres. Nous examinerons divers concepts de programmation orientée objet dans ce livre. Une fois que vous maîtriserez C#, vous serez familiarisé avec ces concepts. Cela vous facilitera la maîtrise d'autres langages de programmation orientés objet à l'avenir.

Prêt à plonger vos orteils dans le monde de la programmation C#? Commençons.

**Note de bas de page:**

La conversion d'un programme C# en langage machine est en fait un peu plus compliquée que cela. Visual Studio Community convertit simplement un programme C# en MIL, qui signifie Microsoft Intermediate Language. Ce code MIL est ensuite converti en langage machine par un système d'exécution virtuel appelé Common Language Runtime. Pour plus d'informations, vous pouvez consulter <https://msdn.microsoft.com/enus/library/z1zx9t92.aspx> . Néanmoins, pour notre objectif, nous n'avons pas besoin de connaître ces détails complexes pour développer nos propres programmes C#.

# **Chapitre 2: Se préparer pour C#**

## **Installation de Visual Studio Community**

Avant de pouvoir commencer à développer des applications en C#, nous devons télécharger Visual Studio Community. Comme mentionné dans le chapitre 1, Visual Studio Community (VSC) est un complément gratuit fourni par Microsoft.

En fait, VSC est plus qu'un simple compilateur. Il s'agit d'un environnement de développement intégré (IDE) qui comprend un éditeur de texte pour que nous écrivions notre code et un débogueur pour nous aider à identifier les erreurs de programmation.

## **Votre premier programme C#**

Pour écrire notre premier programme, créons un dossier sur notre bureau et nommez-le " Projets C#". Nous enregistrerons tous nos projets C# dans ce dossier.

Ensuite, lancez VSC et sélectionnez Fichier> Nouveau> Projet…. (Vous devrez peut-être rechercher « Visual Studio 2015 » si vous ne trouvez pas VSC.) Le premier programme que nous allons écrire est une application console. Les applications de console font référence à des programmes qui n'ont pas d'interface utilisateur graphique.

Dans la boîte de dialogue Nouveau projet, sélectionnez « Visual C#> Windows » (à gauche) et sélectionnez «Application console» dans la boîte principale.

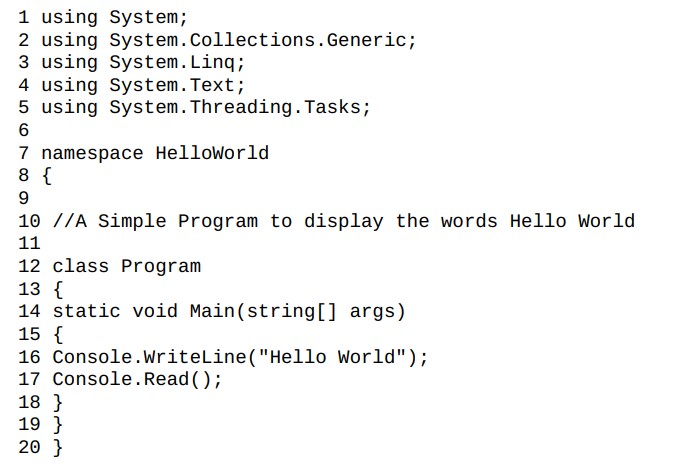
Nommez ce programme «HelloWorld» et enregistrez-le dans le dossier «Projets C#» créé précédemment. Vous pouvez utiliser le bouton « Parcourir… » pour rechercher le dossier approprié.

Enfin, cliquez sur OK pour créer le projet.

Un modèle par défaut créé par VSC vous sera présenté automatiquement.

Remplacez le code du modèle par le code ci-dessous. Notez que les numéros de ligne sont ajoutés pour référence et ne font pas partie du code réel. Vous voudrez peut-être ajouter cette page à vos favoris pour vous y référer plus tard lorsque nous discuterons du programme. En raison de la petite taille de l'écran de la plupart des appareils mobiles, le code peut sembler confus si vous le visualisez sur un Kindle, une tablette ou un téléphone portable. Si vous rencontrez des problèmes pour lire le code, vous pouvez essayer de changer votre écran en mode paysage.

Vous pouvez également télécharger le code source de cet exemple de programme et de tous les autres exemples de programmes de ce livre à l'adresse <http://www.learncodingfast.com/csharp>



Je vous encourage fortement à taper le code vous-même pour avoir une meilleure idée du fonctionnement de VSC. Au fur et à mesure que vous tapez, vous remarquerez qu'une boîte apparaît près du curseur avec des messages d'aide occasionnellement. Cela s'appelle Intellisense. Par exemple, lorsque vous tapez un point (.) Après le mot « Console », une liste déroulante apparaît pour vous indiquer ce que vous pouvez taper après le point. C'est l'une des fonctionnalités de VSC qui facilite le codage pour les programmeurs.

Après avoir fini de taper, vous pouvez exécuter ce programme en cliquant sur le bouton « Démarrer » dans le menu supérieur (voir l'image ci-dessous).

Si votre programme ne s'exécute pas, VSC vous informera de l'erreur dans la « Fenêtre de sortie ». Un double clic sur l'erreur déplace le curseur à l'endroit où se trouve l'erreur.

Vérifiez votre code par rapport au code ci-dessus pour corriger l'erreur et réexécuter le programme.

Si tout se passe bien et que votre programme fonctionne correctement, une fenêtre noire apparaîtra avec les mots «Hello World» en blanc. Cette fenêtre noire est connue sous le nom de console. Appuyez sur Entrée pour fermer la fenêtre.

C'est tout! Vous avez codé avec succès votre premier programme. Donnez-vous une tape sur les épaules.

Si vous accédez maintenant à votre dossier «Projets C#», vous trouverez un dossier nommé Si vous accédez maintenant à votre dossier «Projets C#», vous trouverez un dossier nommé «HelloWorld». Dans le dossier, vous trouverez un autre dossier «HelloWorld» et un fichier «HelloWorld.sln». Ce fichier .sln est le fichier de solution. Chaque fois que vous devez rouvrir un projet, c'est le fichier à ouvrir. Si l'éditeur de texte n'affiche pas votre code lorsque vous ouvrez le fichier de solution, double-cliquez simplement sur le fichier «Program.cs» dans «Explorateur de solutions» à droite (reportez-vous à l'image précédente) pour l'ouvrir.

Le fichier exécutable (.exe) de votre code se trouve dans le dossier HelloWorld> HelloWorld> bin> Debug.

## **Structure de base d'un programme C#**

Maintenant, passons en revue le programme de base que vous venez de coder.

**Directif**

De la ligne 1 à 5, nous avons quelques instructions qui commencent par le mot using. Ces instructions sont appelées directives. Ils indiquent au compilateur que notre programme utilise un certain espace de noms.

Par exemple, la première ligne

using System;

Indique au compilateur que notre programme utilise l'espace de noms System.

**Espace de noms**

Un espace de noms est simplement un regroupement d'éléments de code associés. Ces éléments incluent des classes, des interfaces, des énumérations et des structures, etc. (nous couvrirons chacun de ces éléments dans les chapitres suivants).

C# est livré avec une grande quantité de code pré-écrit organisé en différents espaces de noms. L'espace de noms System contient le code des méthodes qui nous permettent d'interagir avec nos utilisateurs. Nous utilisons deux de ces méthodes dans notre programme

- les méthodes WriteLine() et Read(). Les autres espaces de noms ne sont pas nécessaires dans notre programme. Cependant, comme ces espaces de noms sont inclus dans le modèle par défaut, nous le laisserons dans notre code.

En plus des espaces de noms pré-écrits fournis par Microsoft, nous pouvons également déclarer nos propres espaces de noms.

L'un des avantages de la déclaration d'espaces de noms est qu'elle évite les conflits de noms. Deux ou plusieurs éléments de code peuvent avoir le même nom tant qu'ils appartiennent à des espaces de noms différents. Par exemple, le code ci-dessous définit deux espaces de noms, qui contiennent tous deux une classe nommée MyClass. Ceci est autorisé en C# car les deux classes appartiennent à des espaces de noms différents (First et Second).

namespace First

{

class MyClass

{

}

}

namespace Second

{

Class MyClass

{

}

}

Dans notre exemple, nous avons déclaré un espace de noms - HelloWorld.

L'espace de noms HelloWorld commence à la ligne 7, avec une accolade ouvrante à la ligne 8. Il se termine à la ligne 20 par une accolade fermante. Les accolades sont largement utilisées en C# pour indiquer le début et la fin d'un élément de code. Toutes les accolades ouvrantes en C# doivent être fermées avec une accolade fermante correspondante.

Dans l'espace de noms HelloWorld, nous avons la classe Program qui commence à la ligne 12 et se termine à la ligne 19. Dans la classe Program, nous avons la méthode Main() qui commence à la ligne 14 et se termine à la ligne 18.

## La méthode Main()

La méthode Main() est le point d'entrée de toutes les applications de la console C#. Chaque fois qu'une application console est lancée, la méthode Main() est la première méthode à être appelée.

Dans ce livre, chaque fois qu'il vous est demandé d'essayer un certain segment de code, vous devez créer une nouvelle «application console» et saisir le segment de code donné dans la méthode Main() (entre les accolades). Vous pouvez ensuite exécuter le programme pour tester le code.

Remarquez les mots «string [] args» entre parenthèses de notre méthode Main ()?

Cela signifie que la méthode Main() peut accepter un tableau de chaînes en entrée. Ne vous en faites pas pour le moment. Nous aborderons ces sujets dans les chapitres suivants.

Dans notre exemple, la méthode Main() contient deux lignes de code. La première ligne Console.WriteLine («Hello World»); affiche la ligne «Hello World» (sans les guillemets) à l'écran.

La deuxième ligne

Console.Read ();

attend une touche de l'utilisateur avant de fermer la fenêtre.

Les deux instructions ci-dessus se terminent par un point-virgule. Ceci est courant pour la plupart des instructions en C#. Après l'instruction Console.Read (), nous terminons notre code par trois accolades fermantes pour fermer les accolades ouvrantes précédentes.

C'est tout! Il y a tout ce qu'il y a dans ce programme simple.

### **Commentaires**

Nous en avons couvert pas mal dans ce chapitre. Vous devriez maintenant avoir une compréhension de base de la programmation C# et être raisonnablement à l'aise avec VSC.

Avant de terminer ce chapitre, il reste une chose à apprendre: les commentaires.

Si vous vous référez à notre exemple «HelloWorld» et regardez la ligne 10, vous devriez remarquer que cette ligne commence par deux barres obliques (//).

// Un programme simple pour afficher les mots Hello World

Cette ligne ne fait pas partie du programme. C'est un commentaire que nous écrivons pour rendre notre code plus lisible pour les autres programmeurs. Les commentaires sont ignorés par le compilateur.

Pour ajouter des commentaires à notre programme, nous tapons deux barres obliques (//) devant chaque ligne de commentaire comme ceci

// Ceci est un commentaire

// Ceci est un autre commentaire

// Ceci est encore un autre commentaire

Alternativement, nous pouvons également utiliser / \*… \* / pour des commentaires multi lignes comme celui-ci

/ \* Ceci est un commentaire

C'est aussi un commentaire

Ceci est encore un autre commentaire

\* /

Les commentaires peuvent également être placés après une instruction, comme ceci: Console.Read (); // lit le caractère suivant.

# **Chapitre 3: Le monde des variables et des opérateurs**

Maintenant que vous êtes familiarisé avec VSC et que vous avez écrit votre premier programme, passons aux choses réelles. Dans ce chapitre, vous apprendrez tout sur les variables et les opérateurs. Plus précisément, vous apprendrez ce que sont les variables et comment les nommer, les déclarer et les initialiser. Vous découvrirez également les opérations courantes que nous pouvons effectuer sur eux.

## **Que sont les variables?**

Les variables sont des noms donnés aux données que nous devons stocker et manipuler dans nos programmes. Par exemple, supposons que votre programme ait besoin de stocker l'âge d'un utilisateur.

Pour ce faire, nous pouvons nommer ces données userAge et déclarer la variable userAge à l'aide de l'instruction suivante:

int userAge;

L'instruction de déclaration indique d'abord le type de données de la variable, suivi de son nom. Le type de données d'une variable fait référence au type de données que la variable stockera (par exemple, s'il s'agit d'un nombre ou d'un morceau de texte). Dans notre exemple, le type de données est int, qui fait référence à des entiers. Le nom de notre variable est userAge.

Après avoir déclaré la variable userAge, votre programme allouera une certaine zone de l'espace de stockage de votre ordinateur pour stocker ces données. Vous pouvez alors accéder et modifier ces données en vous y référant par son nom, userAge.

## **Types de données en C#**

Il existe un certain nombre de types de données couramment utilisés en C#.

### **Int (entier)**

int représente un entier (c'est-à-dire des nombres sans partie décimale ou fractionnaire) et contient des nombres de -2 147 483 648 à 2 147 483 647. Les exemples incluent 15; 407; -908; 6150 etc.

### Byte (octet)

byte fait également référence aux nombres entiers, mais a une plage plus étroite de 0 à 255.

La plupart du temps, nous utilisons int au lieu de byte pour les nombres entiers. Cependant, si vous programmez pour une machine qui a un espace mémoire limité, vous devez utiliser byte si vous êtes certain que la valeur de la variable ne dépassera pas la plage de 0 à 255.

Par exemple, si vous avez besoin de stocker l'âge d'un utilisateur, vous pouvez utiliser le type de données byte car il est peu probable que l'âge de l'utilisateur dépasse un jour 255 ans.

### **float (flottant)**

float fait référence aux nombres à virgule flottante, qui sont des nombres avec des décimales telles que 12,43 ; 5,2 et -9,12.

float peut stocker des nombres de -3,4 x 10EX38 à +3,4 x 10EX38. Il utilise 8 octets de stockage et a une précision d'environ 7 chiffres. Cela signifie que si vous utilisez float pour stocker un nombre comme 1,23456789 (10 chiffres), le nombre sera arrondi à 1,234568 (7 chiffres).

### Double

double est également un nombre à virgule flottante, mais peut stocker une plage de nombres beaucoup plus large. Il peut stocker des nombres de (+/-) 5,0 x 10EX-324 à (+/-) 1,7 x 10EX308 et a une précision d'environ 15 à 16 chiffres.

double est le type de données à virgule flottante par défaut en C#. En d'autres termes, si vous écrivez un nombre comme 2.34, C# le traite comme un double par défaut.

### Decimal (décimal)

decimal stocke un nombre décimal mais a une plage plus petite que float et double. Cependant, il a une précision beaucoup plus grande d'environ 28 à 29 chiffres.

Si votre programme nécessite un haut degré de précision lors du stockage de nombres non entiers, vous devez utiliser un type de données décimal. Un exemple est lorsque vous écrivez une application financière où la précision est très importante.

### **Char (caractère)**

char signifie caractère et est utilisé pour stocker des caractères Unicode uniques tels que «A», «%», «@» et «p», etc.

### Bool (booléen)

bool signifie booléen et ne peut contenir que deux valeurs: true (vrai) et false (faux). Il est couramment utilisé dans les instructions de flux de contrôle. Nous aborderons les instructions de flux de contrôle au chapitre 6.

## **Nommer une variable**

Un nom de variable en C# ne peut contenir que des lettres, des chiffres ou des traits de soulignement (\_).

Toutefois, le premier caractère ne peut pas être un chiffre. Vous pouvez donc nommer vos variables userName, user\_name ou userName2 mais pas 2userName.

En outre, il existe des mots réservés que vous ne pouvez pas utiliser comme nom de variable parce qu'ils ont déjà une signification prédéfinie en C#. Ces mots réservés comprennent des mots comme Console, if, while, etc. Nous en apprendrons davantage sur chacun d'entre eux dans les chapitres suivants.

Enfin, les noms de variables sont sensibles à la casse. username n'est pas la même chose que userName.

Il existe deux conventions pour nommer une variable en C#. Nous pouvons soit utiliser la notation en majuscules de chameau, soit utiliser des soulignements. La casse du chameau est la pratique qui consiste à écrire des mots composés avec une casse mixte, en mettant en majuscule la première lettre de chaque mot sauf le premier (par exemple, thisIsAVariableName). C'est la convention que nous utiliserons dans le reste du livre. Une autre pratique courante est d'utiliser le soulignement (\_) pour séparer les mots. Si vous préférez, vous pouvez nommer vos variables comme ceci : this\_is\_a\_variable\_name.

## **Initialisation d'une variable**

Chaque fois que vous déclarez une nouvelle variable, vous devez lui donner une valeur initiale. C'est ce qu'on appelle l'initialisation de la variable. Vous pouvez modifier la valeur de la variable dans votre programme par la suite.

Il y a deux façons d'initialiser une variable. Vous pouvez l'initialiser au moment de la déclaration ou l'initialiser dans une déclaration séparée.

Les exemples ci-dessous montrent comment vous pouvez initialiser une variable au moment de la déclaration :

**Exemple** **1**

Ces exemples montrent comment vous pouvez initialiser les variables byte et int.

byte userAge = 20 ;

int numberOfEmployees = 510 ;

Comme byte et int sont des données sans décimales, vous obtiendrez une erreur si vous écrivez quelque chose comme

byte userAge2 = 20.0 ;

20.0 n'est pas la même chose que 20 en C#.

**Exemple 2**

Les exemples suivants montrent comment vous pouvez initialiser des variables doubles, flottantes et décimales avec des valeurs intégrales. Bien que ces types de données concernent des nombres avec des parties décimales, nous pouvons également les utiliser pour stocker des valeurs intégrales, comme indiqué ci-dessous.

double intNumberOfHours = 5120 ;

float intHourlyRate = 60 ;

decimal intIncome = 25399 ;

**Exemple 3**

Les exemples ci-dessous montrent comment vous pouvez initialiser le double, le flottant et le décimal des variables non entières.

double numberOfHours = 5120.5 ;

float hourlyRate = 60.5f ;

decimal income = 25399.65m ;

Comme mentionné précédemment, le type de données par défaut pour un nombre avec des décimales est double.

Par conséquent, dans les exemples ci-dessus, lorsque vous initialisez le taux horaire (hourlyRate), vous devez ajouter "f" comme suffixe après 60.5 pour indiquer explicitement au compilateur de transformer 60.5 en un flottant.

De même, lorsque vous initialisez la remise (income), vous devez ajouter le suffixe "m" pour transformer 25399.65 en type de données décimales.

**Exemple 4**

Un type de données char ne peut contenir qu'un seul caractère. Lorsque nous initialisons une variable caractère, nous devons mettre ce caractère entre guillemets simples. Voici un exemple :

char grade = 'A' ;

**Exemple 5**

Une variable bool ne peut être que vraie ou fausse. L'exemple ci-dessous montre comment vous pouvez initialiser une variable bool.

bool promote = true ;

**Exemple 6**

En plus d'initialiser les variables individuellement, vous pouvez également initialiser plusieurs dans la même déclaration pour autant qu'elles soient du même type de données. L'exemple suivant montre comment cela peut être fait. Notez que les deux variables sont séparées par une virgule et l'énoncé se termine par un point-virgule.

byte level = 2, userExperience = 5 ;

Les six exemples ci-dessus montrent comment vous pouvez initialiser une variable au point de déclaration. Vous pouvez également choisir d'initialiser une variable dans une déclaration séparée. Un exemple est présenté ci-dessous :

byte year ; //déclarer la variable

year = 20 ; //initialisez-la

## Le signe d'affectation

Le signe = en programmation a une signification différente de celle du signe = que nous avons appris en mathématiques. En programmation, le signe = est connu comme un signe d'affectation. Cela signifie que nous attribuons la valeur du côté droit du signe = à la variable de gauche. Une bonne façon de comprendre un énoncé comme year = 20 est de le considérer comme year <- 20.

En programmation, les énoncés x = y et y = x ont des significations très différentes.

Vous êtes confus ? Un exemple permettra probablement de clarifier ce point.

Supposons que nous ayons deux variables x et y et

x = 5 ; y = 10 ;

Si vous écrivez

x = y ;

votre professeur de mathématiques va probablement vous en vouloir puisque x n'est pas égal à y.

Cependant, en programmation, c'est très bien. Cette déclaration signifie que nous attribuons la valeur de y à x (pensez à x <- y). Il n'y a pas de mal à attribuer la valeur d'une variable à une autre variable. Dans notre exemple, la valeur de x devient 10 alors que la valeur de y reste inchangée. En d'autres termes, x = 10 et y = 10 maintenant.

Supposons maintenant que nous modifions les valeurs de x et de y pour les ramener à

x = 5 ; y = 10 ;

Si vous écrivez maintenant

y = x ;

Cela signifie que vous attribuez la valeur de x à y (considérez que c'est y <- x).

Mathématiquement, x = y et y = x signifient la même chose. Cependant, ce n'est pas le cas en programmation. Ici, la valeur de y passe à 5 tandis que la valeur de x reste inchangée. En d'autres termes, x = 5 et y = 5 maintenant.

## **Opérateurs de base**

Outre l'attribution d'une valeur initiale à une variable ou l'attribution d'une autre variable à celle-ci, nous pouvons également effectuer les opérations mathématiques habituelles sur les variables. Les opérateurs de base en C# comprennent +, -, \*, / et % qui représentent respectivement l'addition, la soustraction, la multiplication, la division et le modulo respectivement.

**Exemple**

Supposons que x = 7, y = 2

Addition : x + y = 9

Soustraction : x - y = 5

Multiplication : x\*y = 14

Division : x/y = 3 (arrondit la réponse à l'entier inférieur le plus proche)

Modulo : x%y = 1 (donne le reste quand 7 est divisé par 2)

Modulo = reste de la division euclidienne

En C#, la division donne une réponse entière si x et y sont tous deux des entiers. Cependant, si x ou y est un nombre non entier, nous obtiendrons une réponse non entière. Par exemple,

7 / 2 = 3

7.0 / 2 = 3.5

7 / 2.0 = 3.5

7.0 / 2.0 = 3.5

Dans le premier cas, lorsqu'un nombre entier est divisé par un autre nombre entier, vous obtenez un nombre entier comme réponse. La partie décimale de la réponse, s'il y en a une, est tronquée. On obtient donc 3 au lieu de 3,5. Dans tous les autres cas, le résultat est un non entier car au moins un des opérandes est un non entier.

## **Plus d'opérateurs d'affectation**

Outre le signe =, il existe quelques autres opérateurs d'affectation en C# (et dans la plupart des langages de programmation). Il s'agit notamment d'opérateurs comme +=, -= et \*=.

Supposons que nous ayons la variable x, avec une valeur initiale de 10. Si nous voulons incrémenter(augmenter) x de 2, nous pouvons écrire

x = x + 2 ;

Le programme va d'abord évaluer l'expression de droite (x + 2) et attribuer la réponse à la gauche. Ainsi, l'expression ci-dessus devient finalement x <- 12.

Au lieu d'écrire x = x + 2, on peut aussi écrire x += 2 pour exprimer la même signification. Le signe += est en fait une abréviation qui combine le signe d'assignation avec l'opérateur d'addition. Ainsi, x += 2 signifie simplement x = x + 2.

De même, si nous voulons faire une soustraction, nous pouvons écrire x = x - 2 ou x -= 2.

La même chose fonctionne pour les 5 opérateurs mentionnés dans la section ci-dessus.

La plupart des langages de programmation ont également les opérateurs ++ et --. L'opérateur ++ est utilisé lorsque vous voulez augmenter la valeur d'une variable de 1. Par exemple, supposons

int x = 2 ;

Si vous écrivez

x++ ;

la valeur de x devient 3.

Il n'est pas nécessaire d'utiliser le signe = lorsque vous utilisez l'opérateur ++. La déclaration x++ ; est équivalente à x = x + 1 ;

L'opérateur ++ peut être placé devant ou derrière le nom de la variable. Cela affecte l'ordre dans lequel les tâches sont effectuées.

Supposons que nous ayons un entier nommé compteur (counter). Si nous écrivons Console.WriteLine(counter++) ;

Le programme imprime d'abord la valeur originale du compteur avant d'incrémenter le compteur de 1. En d'autres termes, il exécute les tâches dans cet ordre

Console.WriteLine(counter) ;

counter = counter + 1 ;

En revanche, si on écrit

Console.WriteLine(++counter) ;

Le programme incrémente d'abord le compteur de 1 avant d'imprimer la nouvelle valeur du compteur. En d'autres termes, il exécute les tâches dans cet ordre

counter = counter + 1 ;

Console.WriteLine(counter) ;

En plus de l'opérateur ++, nous avons aussi l'opérateur -- (deux signes moins).

Cet opérateur diminue la valeur de la variable de 1.

## **Type de conversion en C#**

Parfois, dans notre programme, il est nécessaire de convertir d'un type de données à un autre, comme par exemple d'un double à un int. C'est ce que l'on appelle le "type casting".

Pour convertir un type de données numériques en un autre, il suffit d'ajouter **new data type** (nouveau type de données) devant les données que nous voulons convertir.

Par exemple, nous pouvons transformer un non entier en un entier comme celui-ci :

int x = (int) 20.9 ;

Lorsque nous transformons 20,9 en un nombre entier, la valeur résultante est 20 et non 21. La partie décimale est tronquée après la conversion.

Nous pouvons également convertir un double en un flottant ou une décimale. Rappelons que nous avons mentionné précédemment que tous les non-entiers sont traités comme des doubles par défaut en C# ? Si nous voulons attribuer un nombre comme 20,9 à un flottant ou à une décimale, nous devons ajouter les suffixes "f" et "m" respectivement. Une autre façon de le faire est d'utiliser une conversion, comme ceci :

float num1 = (float) 20.9 ; ou float num1 = 20.9f

decimal num2 = (decimal) 20.9 ; ou decimal num2 = 20.9m

Les valeurs de num1 et num2 seront toutes deux 20,9.

En plus des conversions entre les types numériques, nous pouvons également faire d'autres types de conversions. Nous examinerons certaines de ces conversions dans les chapitres suivants.

# **Chapitre 4 : Tableaux, chaînes et listes**

Dans le chapitre précédent, nous avons couvert certains des types de données de base qui sont couramment utilisés en C#. Outre ces types de données de base, le C# comporte également quelques types de données avancés. Dans ce chapitre, nous allons couvrir trois types de données avancés : les tableaux (array), les chaînes de caractères (string) et les listes (list). En outre, nous allons discuter de la différence entre un type de données de valeur et un type de données de référence.

## **Tableau**

Un tableau est simplement un ensemble de données qui sont normalement liées les unes aux autres.

Supposons que nous voulions stocker les âges de 5 utilisateurs. Au lieu de les stocker sous forme de user1Age, user2Age, user3Age, user4Age et user5Age, nous pouvons les stocker sous forme de tableau.

Un tableau peut être déclaré et initialisé comme suit :

int[] userAge = {21, 22, 23, 24, 25} ;

int indique que cette variable stocke les valeurs entières.

[] indique que la variable est un tableau au lieu d'une variable normale.

userAge est le nom du tableau.

{21, 22, 23, 24, 25} sont les cinq entiers que le tableau stocke.

En plus de déclarer et d'initialiser un tableau au moment de la déclaration, nous pouvons déclarer un tableau d'abord et l'initialiser ensuite. Pour ce faire, nous devons utiliser l’opérateur **new**:

int[] userAge = new int[5] ;

userAge = new [] {21, 22, 23, 24, 25} ;

La première instruction déclare et crée un tableau pour le stockage de 5 entiers.

La deuxième déclaration initialise le tableau.

Les valeurs individuelles du tableau sont accessibles par leurs index, et les index commencent toujours par une valeur de ZERO, et non de 1. C'est une pratique courante dans presque tous les langages de programmation, tels que Python et Java. La première valeur du tableau a un index de 0, la suivante a un index de 1 et ainsi de suite.

Si nous tapons

Console.WriteLine(userAge[0]) ;

La valeur "21" sera affichée à l'écran.

Si nous tapons

userAge [2] = userAge [2] + 20 ;

Le tableau devient {21, 22, 43, 24, 25}. Autrement dit, 20 est ajouté au troisième élément.

### **Propriétés et méthodes des tableaux**

Le C# est doté d'un certain nombre de propriétés et de méthodes utiles que nous pouvons utiliser avec un tableau.

Nous en apprendrons plus sur les propriétés et les méthodes au chapitre 7, lorsque nous parlerons des classes. Pour l'instant, tout ce que nous devons savoir, c'est que pour utiliser une propriété ou une méthode, nous devons utiliser l'opérateur dot (.). Pour utiliser une propriété, nous tapons le nom de la propriété après le point. Pour utiliser une méthode, nous tapons le nom de la méthode après l'opérateur dot, suivi d'une paire de parenthèses ().

#### **Length**

La propriété Length d'un tableau nous indique le nombre d'éléments qu'il contient.

Par exemple, si nous avons

int [] userAge = {21, 22, 26, 32, 40} ;

userAge.Length est égal à 5 car il y a 5 nombres dans le tableau.

#### **Copy()**

La méthode Copy() permet de copier le contenu d'un tableau dans un autre tableau, en commençant par le premier élément.

En C#, une méthode peut avoir de nombreuses variations différentes. Par exemple, la méthode Copy() se décline en quatre variantes différentes. L'exemple ci-dessous présente l'une de ces quatre variantes. Si vous apprenez à utiliser une variante, vous pouvez comprendre comment utiliser les autres méthodes Copy() avec une relative facilité.

Chaque fois que nous utilisons une méthode, nous devons mettre une paire de parenthèses () après le nom de la méthode. Certaines méthodes nécessitent certaines données pour fonctionner. Ces données sont connues sous le nom d'arguments. Nous incluons ces arguments dans la paire de parenthèses. La méthode Copy() nécessite trois arguments.

Supposons vous ayez :

int [] source = {12, 1, 5, -2, 16, 14};

Et :

int [] dest = {1, 2, 3, 4};

Vous pouvez copier les trois premiers éléments de source dans dest en utilisant la déclaration ci-dessous:

Array.Copy(source, dest, 3) ;

Le premier argument est le tableau qui fournit les valeurs à copier. Le second est le tableau dans lequel les valeurs seront copiées. Le dernier argument spécifie le nombre d'éléments à copier.

Dans notre exemple, notre tableau dest devient {12, 1, 5, 4} alors que le tableau source reste inchangé.

#### **Sort()**

La méthode Sort() nous permet de trier nos tableaux. Elle prend un tableau comme argument.

Supposons que vous ayez

int [] numbers = {12, 1, 5, -2, 16, 14} ;

Vous pouvez trier ce tableau en écrivant

Array.Sort(numbers) ;

Le tableau sera trié par ordre croissant. Ainsi, les nombres deviennent {-2, 1, 5,

12, 14, 16}.

#### **IndexOf()**

Nous utilisons la méthode IndexOf() pour déterminer si une certaine valeur existe dans un tableau.

Si elle existe, la méthode renvoie l'index de la première occurrence de cette valeur. Si elle n'existe pas, la méthode renvoie -1.

Par exemple, si vous avez

int [] numbers = {10, 30, 44, 21, 51, 21, 61, 24, 14} ;

Vous pouvez trouver si la valeur 21 existe dans le tableau en écrivant

Array.IndexOf(numbers, 21) ;

La méthode retourne l'indice de la première valeur trouvée, qui est 3 dans ce cas puisque 21 est le quatrième élément du tableau. Vous pouvez ensuite affecter la réponse à une variable comme celle-ci :

int ans = Array.IndexOf(numbers, 21) ;

La valeur de ans est donc 3. Si vous écrivez

ans = Array.IndexOf(numbers, 100) ;

La valeur de ans est -1 car 100 n'existe pas dans le tableau des nombres.

Nous avons couvert dans cette section certaines des méthodes de tableau les plus couramment utilisées.

Pour une liste complète de toutes les méthodes de tableau disponibles en C#, consultez cette page

<https://msdn.microsoft.com/en-us/library/system.array_methods(v=vs.110).aspx>

#### **String**

Ensuite, examinons le type de données string. Un string (chaîne de caractères) est un morceau de texte. Un exemple de chaîne de caractères est le texte "Hello World".

Pour déclarer et initialiser une variable de type chaîne de caractères, vous écrivez

string message = "Hello World" ;

où message est le nom de la variable de la chaîne et "Hello World" est la chaîne qui lui est attribuée. Notez que vous devez mettre la chaîne de caractères entre guillemets ("").

Vous pouvez également attribuer une chaîne vide à une variable, comme ceci :

string anotherMessage = "" ;

Enfin, nous pouvons joindre deux ou plusieurs chaînes de caractères en utilisant le signe de concaténation (+) et les affecter à une variable. Par exemple, on peut écrire

string myName = "Hello World, " + "my name is Jamie" ;

C'est la même chose que

string myName = "Hello World, je m'appelle Jamie" ;

### **Propriétés et méthodes des chaînes de caractères**

Comme les tableaux, les chaînes de caractères ont un certain nombre de propriétés et de méthodes.

#### **Length**

La propriété Length d'une chaîne de caractères nous indique le nombre total de caractères que la chaîne contient.

Pour trouver la longueur de la chaîne "Hello World", on écrit

"Hello World".Length

Nous obtiendrons la valeur 11 car "Hello" et "World" ont tous deux 5 caractères chacun.

Si l'on ajoute l'espace entre les deux mots, on obtient une longueur totale de 11.

#### **Substring()**

La méthode Substring() est utilisée pour extraire une sous-chaîne d'une chaîne plus longue.

Elle nécessite deux arguments. Le premier indique au compilateur l'index de la position de départ à extraire et le second indique au compilateur la longueur.

Supposons que nous déclarions une variable string message et que nous affections la chaîne "Hello

World" à elle.

string message = "Hello World" ;

Nous pouvons ensuite utiliser message pour appeler la méthode Substring() comme indiqué ci-dessous.

string newMessage = message.Substring(2, 5) ;

Substring(2, 5) extrait une sous-chaîne de 5 caractères du message, à partir de l'index 2 (qui est la troisième lettre car les index commencent toujours par 0).

La sous-chaîne résultante est ensuite affectée à newMessage.

newMessage est donc égal à "llo W".

la variable message, en revanche, n'est pas modifié. Il reste comme "Hello World".

#### **Equals()**

Nous pouvons utiliser la méthode Equals() pour comparer si deux chaînes de caractères sont identiques.

Si nous avons deux chaînes de caractères comme indiqué ci-dessous

string firstString = "C'est Jamie" ;

string secondString = "Bonjour" ;

firstString.Equals ("This is Jamie") ;

renvoie vrai alors que

firstString.Equals(secondString) ;

retourne faux car les deux chaînes (firstString et secondString) ne sont pas égales.

#### **Split()**

La méthode Split() divise une chaîne de caractères en sous-chaînes basées sur un tableau de séparateurs définis par l'utilisateur. Après avoir divisé la chaîne, la méthode Split() renvoie un tableau qui contient les sous-chaînes résultantes.

La méthode Split() nécessite deux arguments - un tableau de chaînes qui agissent comme des séparateurs et un second argument pour spécifier si vous voulez supprimer les chaînes vides du résultat.

Supposons que vous vouliez diviser la chaîne "Peter, John ; Andy, ,David" en sous-chaînes,

vous pouvez le faire comme suit (les numéros de ligne sont ajoutés pour référence) :

1 string [] separator = {", ", "; "} ;

2 string names = "Peter, John; Andy, , David" ;

3 string [] subStrings = names.Split(separator, StringSplitOptions.None) ;

Sur la ligne 1, nous déclarons d'abord un tableau de deux chaînes de caractères qui servent de séparateurs. La première chaîne est une virgule suivie d'un espace et la seconde est un point-virgule suivi d'un espace.

Sur la ligne 2, nous attribuons la chaîne que nous voulons séparer à la variable names. Sur la ligne 3, nous utilisons la variable names pour appeler la méthode Split() et assigner son résultat au tableau de sous-chaînes.

Le résultat du code ci-dessus est le tableau suivant

{"Peter", "John", "Andy", "", "David"}

Ce tableau contient une chaîne vide car il y a un espace entre la virgule qui suit "Andy" et la virgule qui précède "David" dans la chaîne originale. Si vous voulez supprimer la chaîne vide du résultat, vous devez changer la ligne 3 en

string [] substrings = names.Split(separator, StringSplitOptions.RemoveEmptyEntries) ;

Le tableau de sous-chaînes devient donc

{"Peter", "John", "Andy", "David"}

Comme d'habitude, nous n'avons couvert qu'un certain nombre des méthodes de chaînes les plus couramment utilisées. Pour une liste complète de toutes les méthodes de chaînes de caractères disponibles en C#, consultez cette page <https://msdn.microsoft.com/enus/library/system.string_methods(v=vs.110).aspx>

## **Les listes**

Examinons maintenant le dernier type de données de ce chapitre - les listes. Une liste stocke des valeurs comme un tableau, mais des éléments peuvent être ajoutés ou supprimés à volonté.

Un tableau ne peut contenir qu'un nombre fixe de valeurs. Si vous déclarez

int [] myArray = new int [10] ;

myArray ne peut contenir que 10 valeurs. Si vous écrivez myArray [10] (qui fait référence à la 11ème valeur puisque l'index du tableau commence à zéro), vous obtiendrez une erreur.

Si vous avez besoin d'une plus grande flexibilité dans votre programme, vous pouvez utiliser une liste.

Pour déclarer une liste d'entiers, on écrit

List<int> userAgeList = new List<int>() ;

userAgeList est le nom de la liste.

List est un mot-clé pour indiquer que vous déclarez une liste.

Le type de données est encadré par des crochets < >.

Vous pouvez choisir d'initialiser la liste au point de déclaration comme ceci

List<int> userAgeList = new List<int> {11, 21, 31, 41} ;

Pour accéder aux éléments individuels d'une liste, nous utilisons la même notation que lorsque nous accédons aux éléments d'un tableau. Par exemple, pour accéder au premier élément, on écrit

userAgeList [0]. Pour accéder au troisième élément, vous écrivez userAgeList[2].

### **Propriétés et méthodes de la liste**

Le type de données de la liste est également accompagné d'un grand nombre de propriétés et de méthodes.

#### **Add()**

Vous pouvez ajouter des membres à une liste en utilisant la méthode Add().

userAgeList.Add(51) ;

userAgeList.Add(61) ;

userAgeList a maintenant 6 membres : {11, 21, 31, 41, 51, 61}.

#### **Count**

Pour connaître le nombre d'éléments de la liste, utilisez la propriété Count.

userAgeList.Count nous donne 6 car il y a 6 éléments dans la liste pour le moment.

#### **Insert()**

Pour ajouter des membres à une position spécifique, utilisez la méthode Insert().

Pour insérer un membre à la 3ème position, vous écrivez

userAgeList.Insert(2, 51) ;

où 2 est l'index et 51 est la valeur que vous voulez insérer.

userAgeList devient maintenant {11, 21, 51, 31, 41, 51, 61}.

#### **Remove()**

Pour supprimer des membres de la liste, utilisez la méthode Remove(). La méthode Remove() prend un argument et supprime la première occurrence de cet argument.

Par exemple, si nous écrivons

userAgeList.Remove(51) ;

userAgeList devient {11, 21, 31, 41, 51, 61}. Seul le premier '51' est supprimé.

#### **RemoveAt()**

Pour retirer un membre à un endroit précis, utilisez la méthode RemoveAt().

Par exemple, pour supprimer le 4ème élément (index 3), vous écrivez

userAgeList.RemoveAt(3) ;

où 3 est l'index de l'élément à supprimer.

userAgeList devient maintenant {11, 21, 31, 51, 61}.

#### **Contains()**

Pour vérifier si une liste contient un certain membre, utilisez la méthode Contains().

Pour vérifier si userAgeList contient '51', on écrit

userAgeList.Contains(51) ;

Nous obtiendrons la vérité comme résultat.

#### Clear()

Pour supprimer tous les éléments d'une liste, utilisez la méthode Clear(). Si nous écrivons

userAgeList.Clear() ;

nous n'aurons plus d'éléments dans la liste.

Pour une liste complète de toutes les méthodes de liste disponibles en C#, consultez cette page

<https://msdn.microsoft.com/en-us/library/s6hkc2c4(v=vs.110).aspx>

## **Type de valeur vs. Type de référence (Type vs. Reference Type)**

Maintenant que nous sommes familiarisés avec les chaînes de caractères, les tableaux et les listes, examinons un concept important concernant les types de données en C#.

Tous les types de données en C# peuvent être classés soit comme un type de valeur, soit comme un type de référence.

Les types de données abordés au chapitre 3 sont des types de valeurs. Les types de données abordés dans ce chapitre sont des types de référence.

Un type de données de valeur est une variable qui stocke ses propres données.

Lorsque nous écrivons

int myNumber = 5 ;

la variable myNumber enregistre la valeur réelle 5.

Un type de référence, en revanche, ne stocke pas les données réelles. Il stocke plutôt une référence aux données. Il ne dit pas au compilateur quelle est la valeur des données ; il indique au compilateur où trouver les données réelles.

Un exemple de type de référence est une chaîne de caractères. Lorsque vous écrivez une déclaration telle que

string message = "Hello" ;

la variable message ne contient pas la chaîne "Hello".

Au lieu de cela, la chaîne "Hello" est créée et stockée ailleurs dans la mémoire de l'ordinateur. La variable message stocke l'adresse de cet emplacement mémoire.

C'est tout ce que nous avons besoin de savoir sur les types de référence pour le moment. Comme il s'agit d'un livre pour débutants, nous n'entrerons pas dans le détail des raisons pour lesquelles les types de référence sont nécessaires. Sachez simplement qu'il existe une différence entre les types de valeurs et les types de référence ; le premier enregistre une valeur tandis que le second enregistre une adresse.

# **Chapitre 5 : Rendre notre programme interactif**

Maintenant que nous avons couvert les bases des variables et des types de données, écrivons un programme qui les utilise. Dans ce chapitre, nous apprendrons à accepter les entrées des utilisateurs, à stocker les données dans une variable et à afficher des messages à nos utilisateurs.

Prêts ?

## **Afficher des messages aux utilisateurs (concaténation, spécificateurs)**

Pour afficher les messages à nos utilisateurs, nous utilisons la méthode Write() ou WriteLine() fournie par C#, disponible dans l'espace de noms du système (namespace).

La différence entre WriteLine() et Write() est que Writeline() déplace le curseur vers la ligne suivante après l'affichage du message alors que Write() ne le fait pas.

Si nous écrivons

Console.WriteLine ("Bonjour") ;

Console.WriteLine("Comment allez-vous ?") ;

Nous aurons

Bonjour

Comment allez-vous ?

Si nous écrivons

Console.Write ("Bonjour") ;

Console.Write("Comment allez-vous ?") ;

Nous aurons

Bonjour comment allez-vous ?

Notez que dans les exemples ci-dessus, nous avons ajouté le mot Console devant le nom de la méthode chaque fois que nous appelons la méthode WriteLine() ou Write(). Cela s'explique par le fait que ces deux méthodes sont des méthodes statiques de la classe Console. Nous parlerons plus en détail des méthodes statiques au chapitre 7.

Si vous trouvez gênant d'ajouter le mot "Console" chaque fois que vous utilisez ces deux méthodes, vous pouvez ajouter la directive

using static System.Console;

au démarrage de votre programme. Si vous faites cela, vous pouvez simplement écrire

WriteLine ("Hello World") ;

au lieu de

Console.WriteLine("Hellow World") ;

Chaque fois que vous utilisez l'une des méthodes statiques de la classe Console. Il s'agit d'une nouvelle fonctionnalité de C# 6 (la dernière version de C#) et n'est disponible que si vous utilisez le dernier IDE (c'est-à-dire Visual Studio 2015). Pour le reste de nos exemples, nous nous en tiendrons à la première méthode de rétrocompatibilité.

Nous avons déjà vu un exemple de la manière dont nous pouvons utiliser la méthode WriteLine() lorsque nous avons écrit le programme "Hello World" au chapitre 2. Examinons maintenant d'autres exemples. Dans les exemples ci-dessous, nous nous concentrerons sur la méthode WriteLine().

La méthode Write() fonctionne exactement de la même manière.

**Exemple 1**

Pour afficher une simple chaîne de caractères, on écrit

Console.WriteLine ("Bonjour, comment allez-vous ?") ;

Affiche

Bonjour, comment allez-vous ?

**Exemple 2**

Pour afficher la valeur d'une variable, nous passons le nom de la variable comme argument.

Par exemple, supposons que nous ayons

int userAge = 30 ;

Nous affichons la valeur de userAge en écrivant

Console.WriteLine(userAge) ;

Sortie

30

Notez que nous ne mettons pas le nom de la variable (userAge) entre guillemets. Si nous écrivons

Console.WriteLine("userAge") ;

On obtiendra

UserAge comme résultat.

**Exemple 3**

Pour combiner deux ou plusieurs chaînes de caractères et les afficher, on utilise la concaténation (+) signe mentionné dans le chapitre précédent.

Par exemple, si nous écrivons

Console.WriteLine ("Hello, " + "how are you ?" + " I love C#.") ;

Nous aurons

Bonjour, comment allez-vous ? J'adore le C#.

**Exemple 4**

Nous pouvons également utiliser le signe de concaténation pour combiner une chaîne et une variable.

Supposons que nous ayons

int results = 79 ;

La déclaration

Console.WriteLine ("Vous avez obtenu " + results + " points pour votre test.") ;

nous donne

Vous avez obtenu 79 points pour votre test.

Là encore, nous ne mettons pas le nom de la variable entre guillemets. Sinon, nous obtiendrons

Vous avez obtenu results pour votre test.

**Exemple 5**

En plus d'utiliser le signe de concaténation pour combiner des chaînes de caractères et des variables, nous pouvons utiliser des placeholders. Supposons que nous ayons

int results = 79 ;

Si nous écrivons

Console.WriteLine("{0} ! Vous avez obtenu {1} points pour votre test", "Bonjour !", results) ;

nous obtiendrons

Bonjour ! Vous avez obtenu 79 points pour votre test.

Dans cet exemple, nous avons passé en trois arguments à la méthode WriteLine(), séparés par des virgules.

Les trois arguments sont

1) “{0} ! Vous avez obtenu {1} points pour votre test."

2) "Bonjour"

3) results

La première est la chaîne de caractères qui sera affichée. Dans la chaîne, les accolades bouclées font office de caractères de remplissage et seront remplacées par les arguments qui suivent.

{0} est un caractère de remplacement pour l'argument suivant, qui est la chaîne "Bonjour !"  dans ce cas.

{1} est un caractère de remplacement pour "la variable results"

Par conséquent, l’affichage est

Bonjour ! Vous avez obtenu 79 points pour votre test.

Si vous avez écrit

Console.WriteLine("{1} ! Vous avez obtenu {0} points pour votre test", "Bonjour", results) ;

Vous obtiendrez

79 ! Vous avez obtenu bonjour points pour votre test.

Bien sûr, une telle affirmation n'a aucun sens. Cependant, elle montre comment les placeholders sont remplacés par les arguments correspondants.

Nous pouvons spécifier comment nous voulons que les valeurs numériques soient affichées lorsque nous utilisons des caractères de remplissage. Cela se fait à l'aide d'un spécificateur de format, comme les spécificateurs C et F.

Le spécificateur F précise le nombre de décimales avec lesquelles un nombre doit être affiché.

Si nous écrivons

Console.WriteLine("Le numéro est {0:F3}.", 123.45678) ;

On obtiendra

Le numéro est le 123.457.

Le spécificateur F3 arrondit le nombre 123.45678 à 123.457. Notez qu'il ne doit pas être précédé d'un espace. En d'autres termes, il doit être {0:F3} et non {0 : F3}.

Le spécificateur C sert à formater les devises ; il ajoute le symbole "$ ou €" devant le nombre et affiche le nombre avec 2 décimales. En outre, il sépare tous les milliers par une virgule.

Si vous écrivez

Console.WriteLine("Dépôt = {0:C}. Solde du compte = {1:C}.", 2125, 12345.678) ;

Vous obtiendrez

Dépôt = 2 125,00 $. Solde du compte = 12 345,68 $.

**Exemple 6**

Nous pouvons également utiliser Console.WriteLine() pour afficher le résultat d'une méthode.

Au chapitre 4, nous avons appris à utiliser la méthode Substring() pour extraire une sous-chaîne d'une chaîne plus longue. Dans cet exemple, nous avons assigné le résultat à une autre chaîne. Nous pouvons également utiliser la méthode Console.WriteLine() pour afficher le résultat sans lui attribuer de variable.

Par exemple, si vous écrivez

Console.WriteLine("Microsoft".Substring(1, 3)) ;

Sortie **icr** sera affiché à l'écran.

Outre l'affichage du résultat d'une méthode, Console.WriteLine() peut également être utilisé pour afficher la taille d’une chaine de caractère. Si l'on écrit

Console.WriteLine("Hello World".Length) ;

la valeur 11 sera affiché à l'écran.

## **Séquences d'évasion (Escape Sequences : tabulation, saut à la ligne)**

Parfois, dans nos programmes, nous devons afficher certains caractères spéciaux "non affichables" tels qu'une tabulation ou un saut de ligne. Dans ce cas, vous devez utiliser le caractère \ (barre oblique inversée) pour échapper aux caractères qui, autrement, ont une signification différente.

Par exemple, pour afficher une tabulation, nous tapons le caractère backslash avant la lettre t, comme ceci : \t.

Sans le caractère \, la lettre "t" sera affichée. Avec ce caractère, une tabulation est affichée.

Par conséquent, si vous tapez

Console.WriteLine("Hello\tWorld") ;

vous obtiendrez

Hello Worl

Le caractère backslash est également utilisé couramment :

Pour afficher une nouvelle ligne (\n)

**Exemple**

Console.WriteLine("Hello\nWorld") ;

Sortie

Hello

World

Pour afficher le caractère backslash lui-même (\\)

**Exemple**

Console.WriteLine ("\\") ;

Sortie

\

Pour afficher des guillemets doubles (\") afin que le guillemet double ne termine pas la chaîne

**Exemple**

Console.WriteLine(“I am 5’9\” tall”);

Sortie

I am 5’9” tall (Je mesure 1,80 m)

## **Accepter les données de l'utilisateur**

Maintenant que nous savons comment afficher des messages à nos utilisateurs, voyons comment nous pouvons accepter leur contribution.

Pour accepter les entrées des utilisateurs, nous pouvons utiliser la méthode Read() ou ReadLine().

Read() lit le caractère suivant de l'entrée standard tandis que ReadLine() lit une ligne de caractères. L'entrée standard fait référence au dispositif standard que les utilisateurs utilisent pour entrer des données, qui est généralement le clavier.

L'exemple ci-dessous montre comment nous pouvons utiliser la méthode ReadLine() pour lire les données saisies par les utilisateurs. La méthode Read() fonctionne de la même manière.

string userInput = Console.ReadLine() ;

Les méthodes Read() et ReadLine() lisent toutes deux les entrées de l'utilisateur sous la forme d'une chaîne de caractères. Par conséquent, dans l'exemple ci-dessus, nous attribuons le résultat de Console.ReadLine() à une variable de chaîne appelée userInput.

Nous pouvons ensuite utiliser

Console.WriteLine(userInput) ;

Pour afficher la saisie que l'utilisateur a effectuée.

Conversion d'une chaîne de caractères en un nombre (string en nombre)

Parfois, il est nécessaire de convertir les données saisies par les utilisateurs en un type de données numériques afin de pouvoir effectuer des calculs sur celles-ci. C# nous fournit un certain nombre de méthodes pour effectuer la conversion. Les méthodes que nous utilisons se trouvent dans la classe Convert, qui est également regroupée sous l'espace de noms System.

Pour convertir une chaîne de caractères en un entier, nous utilisons la méthode ToInt32(). Par exemple, si nous avons

string userInput = Console.ReadLine() ;

Et la valeur entrée par l’utilisateur est 20, userInput sera égal à "20" (qui est une chaîne et non un nombre entier à cause des guillemets).

Nous pouvons alors utiliser

int newUserInput = Convert.ToInt32(userInput) ;

Pour convertir la chaîne en entier 20 et l'affecter à une variable int. Nous pouvons maintenant effectuer les opérations mathématiques habituelles sur cette nouvelle variable int.

Outre la conversion d'une chaîne de caractères en un nombre entier, nous pouvons également convertir une chaîne de caractères en adécimal, en flottant ou en double en utilisant respectivement les méthodes ToDecimal(), ToSingle() et ToDouble().

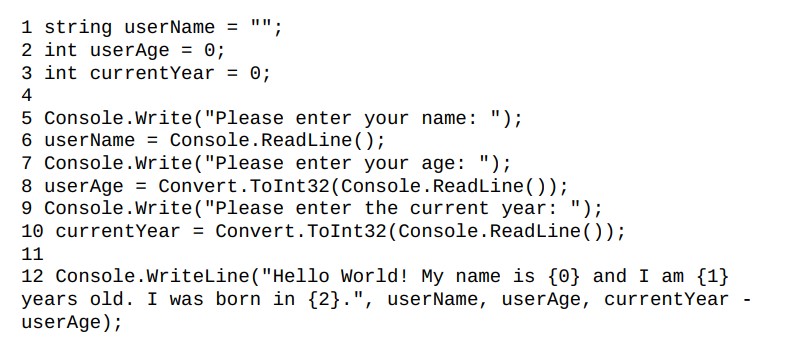
## **Tout mettre ensemble**

Mettons maintenant tout ce que nous avons appris ensemble pour écrire un programme complet. Nous allons modifier le programme "Hello World" que nous avons écrit au chapitre 2.

Au lieu de nous contenter de dire bonjour le monde, nous voulons que le monde connaisse aussi nos noms et nos âges.

Tout d'abord, lancez Visual Studio Community et créez un nouveau projet d'application de console Visual C#. Nommez le projet "HelloWorldAgain".

Tapez le segment de code suivant dans la méthode Main() (les numéros de ligne sont ajoutés pour référence).



Lancez le programme et entrez les informations suivantes

Veuillez entrer votre nom : Jamie

Veuillez indiquer votre âge : 44 ans

Veuillez entrer l'année en cours : 2020

Le programme devrait vous donner les résultats suivants

Bonjour le monde ! Je m'appelle Jamie et j'ai 44 ans. Je suis né en 1976.

Ce programme devrait être assez facile à comprendre. Cependant, il y a deux points à mentionner à propos du programme.

Premièrement, la ligne 10 montre un exemple de la façon dont nous pouvons utiliser deux méthodes au sein d'une même déclaration. Lorsque nous écrivons

userAge = Convert.ToInt32(Console.ReadLine()) ;

La méthode Console.ReadLine() est exécutée en premier car elle se trouve dans une paire de parenthèses (). Ceci est similaire à la façon dont les opérations entre parenthèses ont un ordre de priorité plus élevé lorsque nous évaluons une expression mathématique. Par exemple, lorsque nous évaluons 3 (5 + 9), nous devons d'abord additionner 5 à 9 avant de multiplier la réponse par 3 (c'est-à-dire 3(14)).

Après l'exécution de Console.ReadLine(), la valeur saisie par l'utilisateur est convertie en un nombre entier à l'aide de Convert.ToInt32().

Supposons que l'utilisateur ait entré 44.

Convert.ToInt32(Console.ReadLine())

devient

Convert.ToInt32("44").

Le résultat de Convert.ToInt32("44") est l'entier 44. Cet entier est ensuite affecté à la variable userAge.

La prochaine chose à signaler à propos du programme est la ligne 12, comme indiqué ci-dessous :

Console.WriteLine("Hello World ! Mon nom est {0} et j'ai {1} ans. Je suis né en {2}", userName, userAge, currentYear - userAge) ;

Remarquez que le dernier argument (currentYear - userAge) implique une opération mathématique ? Ceci est autorisé en C#. WriteLine() effectuera la soustraction et affichera le résultat du calcul.

# **Chapitre 6 : Faire des choix et prendre des décisions**

Félicitations pour avoir réussi jusqu'ici. Nous avons parcouru un long chemin. Vous connaissez maintenant les différents types de données en C# et êtes capable de coder un programme simple qui interagit avec les utilisateurs.

Dans ce chapitre, nous allons aborder un autre concept fondamental de la programmation ; nous apprendrons comment contrôler le flux d'un programme à l'aide d'instructions de contrôle.

Plus précisément, nous apprendrons l'instruction if, l'instruction inline if, l'instruction switch, la boucle for, la boucle foreach, la boucle while et la boucle do while. De plus, nous apprendrons également à propos de la déclaration try-catch-finally qui contrôle le flux du programme lorsqu'une erreur se produit.

Cependant, avant d'aborder ces outils de contrôle, nous devons d'abord examiner les déclarations de condition.

## **Déclarations de condition**

La plupart des énoncés de flux de contrôle impliquent l'évaluation d'un énoncé de condition. Le programme se déroulera différemment selon que la condition est remplie ou non.

#### **Egalité (==)**

La déclaration de condition la plus courante est la déclaration de comparaison. Si nous voulons comparer si deux variables sont identiques, nous utilisons le signe == (double =).

Par exemple, si vous écrivez x == y, vous demandez au programme de vérifier si la valeur de x est égale à la valeur de y. Si elles sont égales, la condition est remplie et l'énoncé est évalué comme vrai. Dans le cas contraire, la déclaration est considérée comme fausse.

En plus d'évaluer si une valeur est égale à une autre, il existe d'autres opérateurs de comparaison que nous pouvons utiliser dans nos déclarations de conditions.

#### **Non égal ou différent (!=)**

Retourne vrai si la gauche n'est pas égale à la droite

5 != 2 est vrai

6 != 6 est faux

#### **Plus grand que (>)**

Retourne vrai si la gauche est plus grande que la droite

5 > 2 est vrai

3 > 6 est faux

#### **Plus petit que (<)**

Retourne vrai si la gauche est plus petite que la droite

1 < 7 est vrai

9 < 6 est faux

#### **Supérieur ou égal à (>=)**

Retourne vrai si la gauche est supérieure ou égale à la droite

5 >= 2 est vrai

5 >= 5 est vrai

3 >= 6 est faux

#### Inférieur ou égal à (<=)

Retourne vrai si la gauche est inférieure ou égale à la droite

3 <= 7 est vrai

7 <= 7 est vrai

9 <= 6 est faux

Nous avons également trois opérateurs logiques (&&, ||, !=) qui sont utiles si nous voulons combiner plusieurs conditions.

#### **L'opérateur AND (&&)**

Retourne vrai si toutes les conditions sont remplies

5==5 && 2>1 && 3!=7 est vrai

5==5 && 2<1 && 3!=7 est faux car la deuxième condition (2<1) est fausse

#### **L'opérateur OR (||)**

Il est considéré comme vrai si au moins une condition est remplie.

5==5 || 2<1 || 3==7 est vrai car la première condition (5==5) est vraie

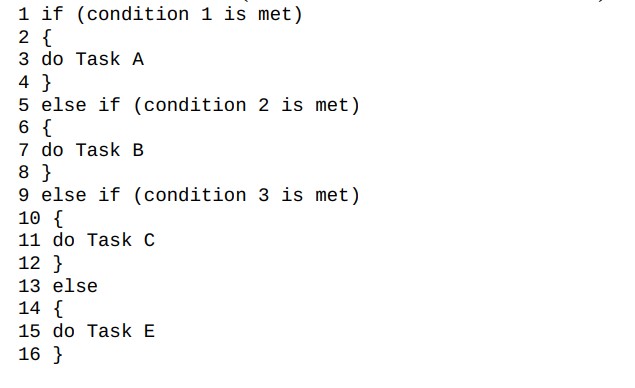
5==6 || 2<1 || 3==7 est faux car toutes les conditions sont fausses

## **Contrôler les déclarations de flux ou de condition**

Maintenant que nous sommes familiarisés avec les déclarations de conditions, apprenons comment nous pouvons utiliser ces déclarations pour contrôler le déroulement d'un programme.

### **La déclaration If (Si)**

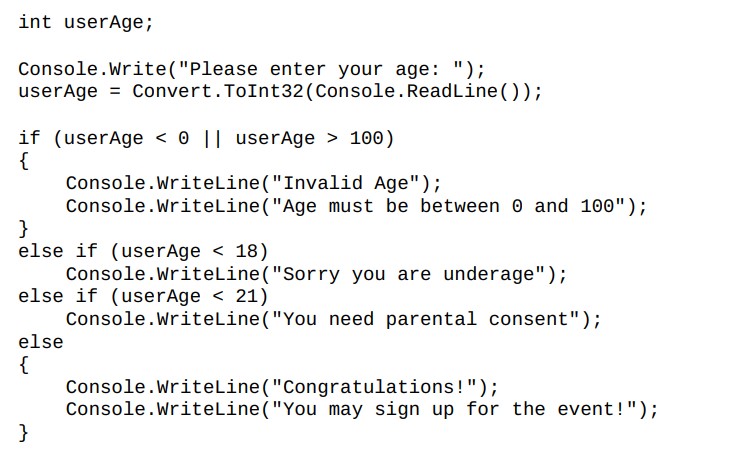
La déclaration if est l'une des déclarations de flux de contrôle les plus couramment utilisées. Elle permet au programme d'évaluer si une certaine condition est remplie, et d'effectuer l'action appropriée en fonction du résultat de l'évaluation. La structure d'une déclaration "if" est la suivante (les numéros de ligne sont ajoutés pour référence) :



La ligne 1 teste la première condition. Si la condition est remplie, tout ce qui se trouve à l'intérieur de la paire de bretelles bouclées {} qui suit (lignes 2 à 4) sera exécuté. Le reste de l'énoncé if (de la ligne 5 à 16) sera ignoré.

Si la première condition n'est pas remplie, vous pouvez utiliser les autres déclarations if qui suivent pour tester d'autres conditions (lignes 5 à 12). Il peut y avoir plusieurs instructions if. Enfin, vous pouvez utiliser l'instruction else (lignes 13 à 16) pour exécuter du code si aucune des conditions précédentes n'est remplie.

Pour bien comprendre le fonctionnement de l'instruction if, ajoutez le code suivant au programmeMain() du modèle VSC.



Le programme demande d'abord à l'utilisateur son âge et enregistre le résultat dans la variable userAge.

Ensuite, la déclaration

si (userAge < 0 || userAge > 100)

Vérifie si la valeur de userAge est inférieure à zéro ou supérieure à 100. Si l'une des conditions est vraie, le programme exécute toutes les déclarations dans les accolades qui suivent. Dans cet exemple, il affichera "Invalid Age", suivi de

"Age must be between 0 and 100".

D'autre part, si les deux conditions sont fausses, le programme testera la condition suivante – else if (userAge < 18). Si userAge est inférieur à 18 ans (mais supérieur ou égal à 0 puisque la première condition n'est pas remplie), le programme affichera " Sorry you are underage ".

Vous remarquerez peut-être que nous n'avons pas joint la déclaration

Console.WriteLine ("Sorry you are underage ");

Dans les {}. En effet, les accolades sont facultatives s'il n'y a qu'une seule déclaration à exécuter.

Si l'utilisateur n'a pas saisi une valeur inférieure à 18, mais a saisi une valeur supérieure ou égale à 18 mais inférieure à 21, l'instruction suivante sera exécutée. Dans ce cas, le message " You need parental consent" sera affiché.

Enfin, si l'utilisateur a saisi une valeur supérieure ou égale à 21 mais inférieure ou égale à 100, le programme exécutera le code dans le bloc "else". Dans ce cas, il affichera le message " Congratulations" suivi de " You may sign up for the event!".

Exécutez le programme cinq fois et entrez respectivement -1, 8, 20, 23 et 121 pour chaque exécution. Vous obtiendrez les résultats suivants :

Veuillez entrer votre âge : -1

Âge invalide

L'âge doit être compris entre 0 et 100 ans

Veuillez indiquer votre âge : 8

Désolé, vous êtes mineur

Veuillez indiquer votre âge : 20 ans

Vous devez obtenir l'autorisation des parents

Veuillez indiquer votre âge : 23 ans

Félicitations !

Vous pouvez vous inscrire à l'événement !

Veuillez indiquer votre âge : 121

Âge invalide

L'âge doit être compris entre 0 et 100 ans.

### **Inline If (Si en ligne appelé aussi ternaire)**

Les **ternaires** permettent, sur une seule ligne, de changer la valeur d'une variable en fonction d'une condition.

Une déclaration if en ligne est une forme plus simple de déclaration if qui est très pratique si vous voulez attribuer une valeur à une variable en fonction du résultat d'une condition. La syntaxe est :

condition ? valeur si la condition est vraie : valeur si la condition est fausse ;

Par exemple, l'instruction

3>2 ? 10 : 5 ;

renvoie la valeur 10 puisque 3 est supérieur à 2 (c'est-à-dire que la condition 3 > 2 est vraie).

Cette valeur peut ensuite être attribuée à une variable.

Si nous écrivons

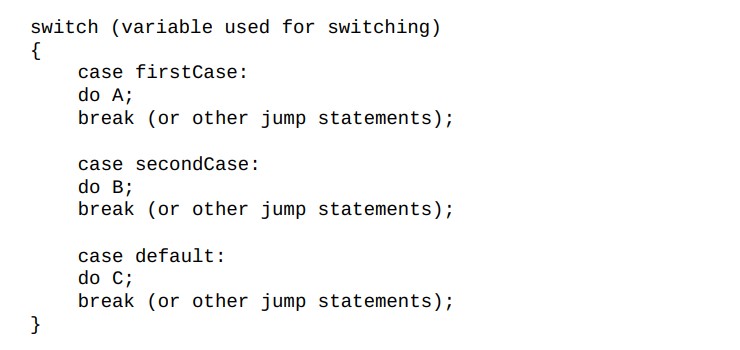
int myNum = 3>2 ? 10 : 5 ;

myNum se verra attribuer la valeur 10.

### **Switch (déclaration de commutation)**

La déclaration switch est similaire à une déclaration if, sauf qu'elle ne fonctionne pas avec une plage de valeurs. Une déclaration switch exige que chaque cas soit basé sur une seule valeur. En fonction de la valeur de la variable utilisée pour la commutation, le programme exécutera le bloc de code correct.

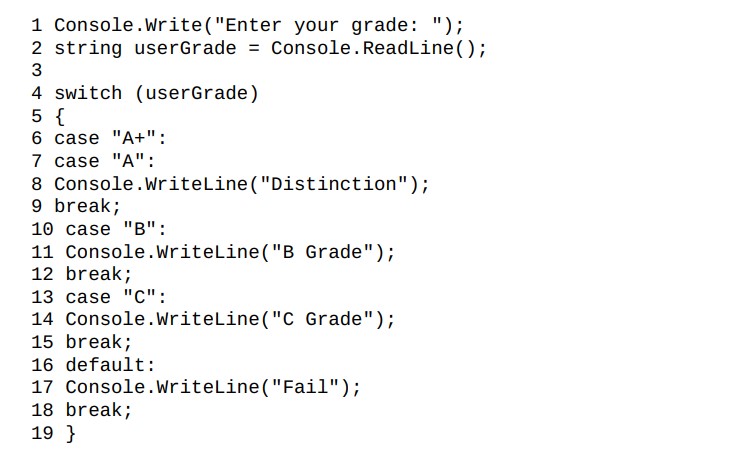
La syntaxe d'une instruction switch est la suivante :



Vous pouvez avoir autant de cas que vous le souhaitez lorsque vous utilisez une instruction switch. Le cas par défaut est facultatif et est exécuté si aucun autre cas ne s'applique.

Lorsqu'un certain cas est satisfait, tout ce qui commence à la ligne suivante est exécuté jusqu'à ce qu'une instruction jump soit atteinte. Une instruction de saut est une instruction qui ordonne au compilateur de passer à une autre ligne du programme. Nous examinerons plus en détail les instructions de saut plus tard. L'instruction de saut la plus couramment utilisée est l'instruction break ;.

Voyons un exemple de fonctionnement de la déclaration de commutation.



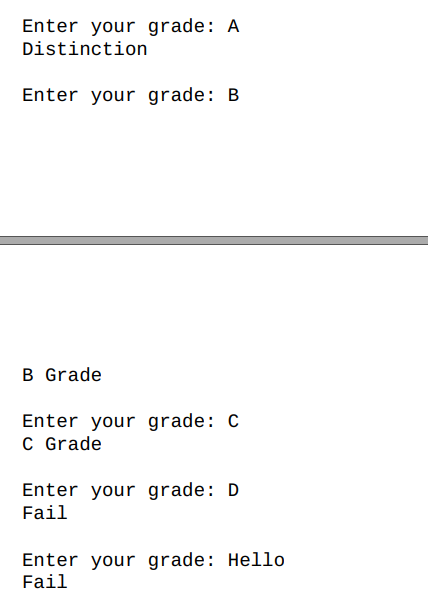
Le programme invite d'abord l'utilisateur à indiquer sa note.

Si la note est "A+" (ligne 6), le programme exécute l'énoncé suivant jusqu'à ce qu'il atteigne la pause ; énoncé. Cela signifie qu'il exécutera les lignes 7 à 9. La sortie est donc "Distinction".

Si la note est "A" (Ligne 7), le programme exécute les Lignes 8 et 9. De même, la sortie est "Distinction".

Si la note n'est pas "A+" ou "A", le programme vérifie le cas suivant. Il continue à vérifier de haut en bas jusqu'à ce qu'un cas soit satisfait. Si aucun des cas ne s'applique, le cas par défaut est exécuté.

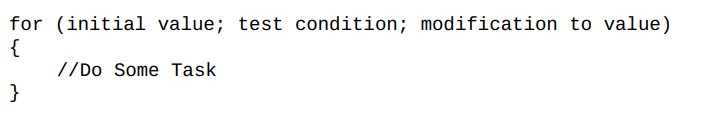
Si vous exécutez le code ci-dessus, vous obtiendrez la sortie suivante pour chacune des entrées indiquées :



### **La boucle Pour (For Loop)**

La boucle for exécute un bloc de code de manière répétée jusqu'à ce que la condition de test ne soit plus valable.

La syntaxe de la boucle for est la suivante :

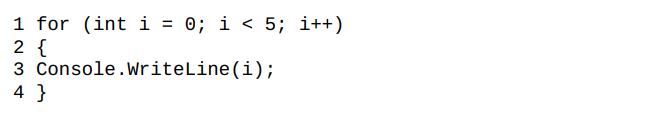


Pour (valeur initiale ; condition de test; modification de la valeur)

{

// Faire ceci

}

Pour comprendre comment fonctionne la boucle for, considérons l'exemple ci-dessous.

Le point central de la boucle for est la ligne 1 :

Elle se compose de trois parties, chacune étant séparée par un point-virgule.

La première partie déclare et initialise une variable int i à zéro. Cette variable sert de compteur de boucle.

La deuxième partie vérifie si i est inférieur à 5. Si c'est le cas, les déclarations à l'intérieur des accolades seront exécutées. Dans cet exemple, les accolades sont facultatives car il n'y a qu'une seule déclaration.

Après avoir exécuté l'instruction WriteLine(), le programme retourne au dernier segment de la ligne 1. i++ incrémente la valeur de i de 1. Ainsi, i passe de 0 à 1.

Après l'incrémentation, le programme teste si la nouvelle valeur de i est toujours inférieure à 5. Si c'est le cas, il exécute à nouveau l'instruction WriteLine().

Ce processus de test et d'incrémentation du compteur de boucle est répété jusqu'à ce que la condition i < 5 n'est plus vraie. À ce stade, le programme quitte la boucle for et continue à exécuter d'autres commandes après la boucle for.

La sortie pour le segment de code est :



La sortie s'arrête à 4 car lorsque i est égal à 5, l'instruction WriteLine() n'est pas exécutée car 5 n'est pas inférieur à 5.

La boucle for est couramment utilisée pour boucler un tableau ou une liste. Par exemple, si nous avons

**int [] myNumbers = {10, 20, 30, 40, 50} ;**

Nous pouvons utiliser une boucle for et la propriété Length du tableau pour faire une boucle à travers le tableau comme indiqué ci-dessous.

**for (int i = 0 ; i < myNumbers.Length ; i++)**

**{**

**Console.WriteLine(myNumbers[i] ;**

**}**

Comme myNumbers.Length est égal à 5, ce code va de i = 0 à i = 4. Si nous exécutons le code, nous obtiendrons la sortie suivante :



### **La boucle foreach (foreach loop)**

En plus des boucles for, nous pouvons également utiliser une boucle foreach lorsque nous travaillons avec des tableaux et des listes. Une boucle foreach est très utile si vous voulez obtenir des informations à partir d'un tableau ou d'une liste, sans y apporter de modifications.

Supposons que vous ayez

**char [] message = { ‘H’, ‘e’, ‘l’, ‘l, ‘o’ }**

Vous pouvez utiliser le code suivant pour afficher les éléments du tableau.

**foreach (char i in message)**

**Console.Write(i);**

Dans le code ci-dessus, nous avons une variable char i qui est utilisée pour le bouclage. Chaque fois que la boucle se déroule, un élément du tableau message est attribué à la variable i. Par exemple, la première fois que la boucle se déroule, le caractère "H" est attribué à i.

La ligne

**Console.WriteLine(i) ;**

Affiche la lettre ‘H’

La deuxième fois que la boucle se déroule, le caractère "e" est attribué à i. La ligne

**Console.Write(i) ;**

affiche la lettre "e".

Cette opération se poursuit jusqu'à ce que tous les éléments du tableau soient affichés.

### **La boucle Tant que (While loop)**

Comme son nom l'indique, la boucle while exécute à plusieurs reprises des instructions à l'intérieur de la boucle alors qu'une certaine condition reste valable. La structure de l'instruction while est la suivante :

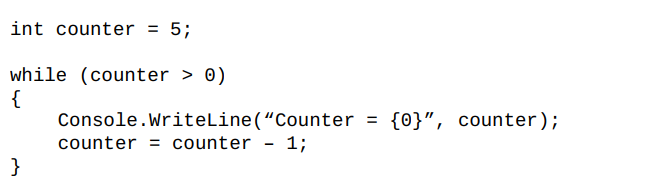
While (condition is true)

{

do A

}

La plupart du temps, lorsque l'on utilise une boucle while, il faut d'abord déclarer une variable pour qu’elle fonctionne comme compteur de boucle. Appelons cette variable counter. Le code ci-dessous montre un exemple de fonctionnement d'une boucle while.



Si vous exécutez le code, vous obtiendrez la sortie suivante :



La syntaxe d'une déclaration "while" est relativement simple. Les énoncés à l'intérieur des accolades de la boucle sont exécutées tant que la variable counter > 0.

Remarquez que nous avons la ligne counter = cunter - 1 à l'intérieur des accolades ?

Cette ligne est cruciale. Elle diminue la valeur du compteur de 1 chaque fois que la boucle est exécutée.

Nous devons diminuer la valeur de counter de 1 pour que la condition de la boucle (compteur > 0) soit finalement évaluée comme fausse. Si nous oublions de faire cela, la boucle continuera à tourner sans fin, ce qui donnera une boucle infinie. Le programme continuera d’afficher counter = 5 jusqu'à ce que vous le tuiez d'une manière ou d'une autre. Ce n'est pas une expérience agréable, surtout si vous avez un gros programme et que vous n'avez aucune idée du segment de code qui provoque la boucle infinie.

### **Faire Tant que (Do while)**

La boucle "do while" est similaire à la boucle "while" avec une différence principale : le code dans les accolades d'une boucle "do while" est exécuté au moins une fois. Voici un exemple de fonctionnement d'une boucle do while.

Int counter = 100 ;

do {

Console.WriteLine("Compteur = {0}, counter) ;

counter++ ;

} while (counter < 0);

Comme la condition de test (while (counter < 0)) est placée après la fermeture de l'accolade, elle est testée après que le code à l'intérieur des accolades ait été exécuté au moins une fois.

Si vous exécutez le code ci-dessus, vous obtiendrez

counter = 100 ;

Après la première exécution de l'instruction WriteLine(), le compteur est incrémenté de 1. La valeur du compteur est maintenant de 101. Lorsque le programme atteint la condition de test, le test échoue car la valeur du compteur n'est pas inférieure à 0. Le programme quitte alors la boucle. Même si la valeur originale du compteur ne remplit pas la condition de test (compteur < 0), le code à l'intérieur des accolades est toujours exécuté une fois.

Notez que pour une déclaration "do while", un point-virgule ( ;) est nécessaire après la condition de test.

### **Instructions de saut (Jump Statements)**

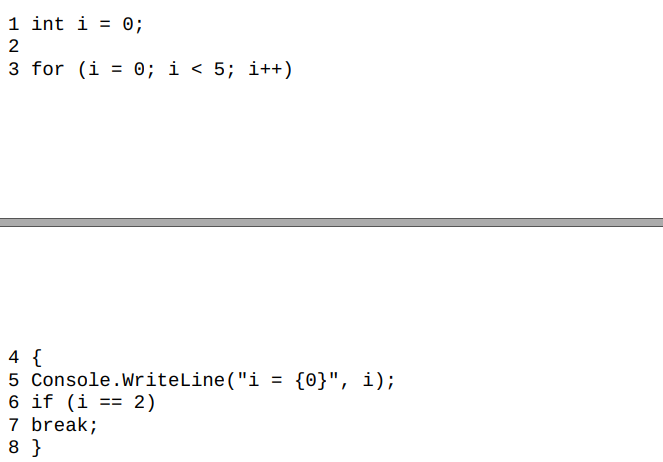
Nous avons maintenant couvert la plupart des instructions de flux de contrôle en C#. Ensuite, examinons les instructions de saut.

Une instruction de saut est une instruction qui ordonne au programme de s'écarter de sa séquence de flux normale et de sauter à une autre ligne de code. Les instructions de saut sont couramment utilisées dans les boucles et autres instructions de flux de contrôle.

#### **Break (Pause)**

Le mot-clé break fait que le programme sort prématurément d'une boucle lorsqu'une certaine condition est remplie. Nous avons déjà vu comment le mot-clé break peut être utilisé dans une instruction de commutation (switch). Voyons maintenant un exemple de la façon dont le mot-clé break peut être utilisé dans une boucle for.

Considérons le segment de code ci-dessous :



Dans cet exemple, nous avons utilisé une déclaration if à l'intérieur d'une boucle for. Il est très courant pour nous de "mélanger" différents outils de contrôle en programmation, comme l'utilisation d'une boucle while dans une instruction if ou l'utilisation d'une boucle for dans une boucle while. C'est ce qu'on appelle une **instruction de contrôle imbriquée**.

Si vous exécutez le segment de code ci-dessus, vous obtiendrez la sortie suivante.

i = 0

i = 1

i = 2

Remarquez que la boucle se termine prématurément à i = 2 ?

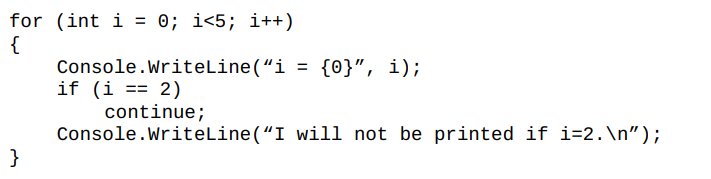
Sans le mot-clé break, la boucle devrait se dérouler de i = 0 à i = 4 car la condition de la boucle est i < 5. Cependant, avec le mot-clé break, lorsque i = 2, la condition de la ligne 6 s'évalue à vrai. Le mot-clé break de la ligne 7 provoque alors la fin prématurée de la boucle.

#### **Continue**

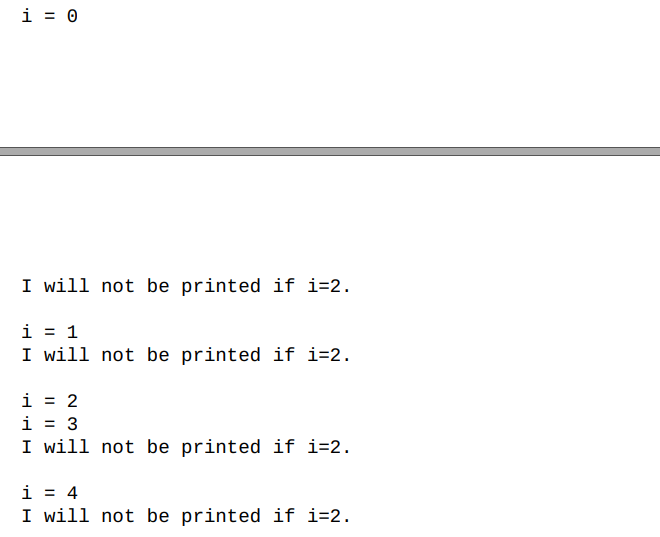
Un autre mot-clé couramment utilisé pour les sauts est le mot-clé continue. Lorsque nous utilisons

continue, le reste de la boucle après le mot-clé est sauté pour cette itération. Un exemple le rendra plus clair.

Si vous exécutez le segment de code ci-dessous



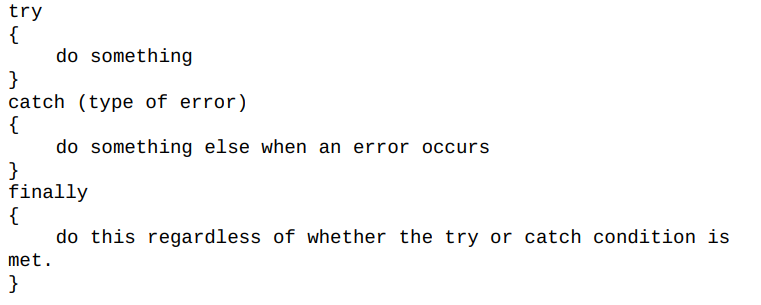
Vous obtiendrez le résultat suivant :



Lorsque i = 2, la ligne qui suit le mot-clé continue n'est pas exécutée. A part cela, tout se déroule normalement.

### **Traitement des exceptions(try-catch-finally)**

Nous avons maintenant appris comment contrôler le déroulement d'un programme dans des circonstances "normales" en utilisant des instructions de contrôle et des instructions de saut. Avant de terminer ce chapitre, nous devons examiner une dernière déclaration de contrôle, la déclaration "try-catch-finally". La déclaration "try-catch-finally" contrôle la façon dont le programme se déroule lorsqu'une erreur se produit. La syntaxe est la suivante :



**try**

**{**

**Faire ceci**

**}**

**Catch (type d’erreur)**

**{**

**Faire cela lorsqu'une erreur se produit**

**}**

**Finally**

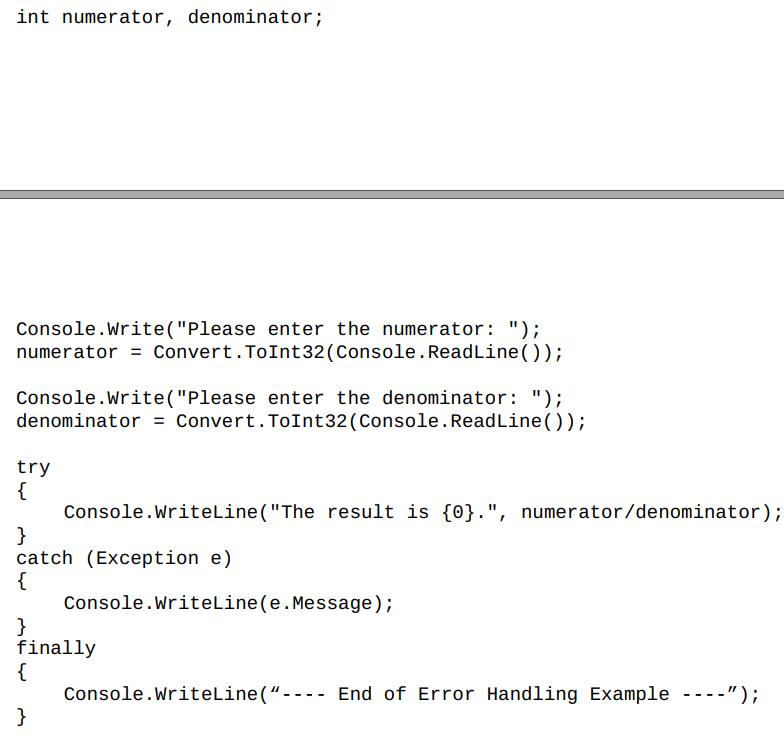
**{**

**Le faire indépendamment du fait que la condition try ou catch soit remplie.**

**}**

Vous pouvez avoir plus d'un bloc de prises. En outre, le dernier bloc est facultatif.

Prenons un exemple.



Si vous lancez le code et que vous entrez 12 et 4, vous obtiendrez le message :

**The result is 3.  
---- End of Error Handling Example ----**

Dans cet exemple, le code dans le bloc try s'exécute avec succès. Après l'exécution du code dans le bloc d'essai, le code dans le bloc final est exécuté.

Supposons maintenant que vous entriez 12 et 0 à la place. Vous obtiendrez

**Attempted to divide by zero.  
---- End of Error Handling Example ----**

Dans ce cas, le code du bloc **catch** est exécuté à la place. En effet, lorsque le programme tente d'exécuter l'instruction du bloc **try**, une erreur se produit puisque vous ne pouvez pas diviser un nombre par zéro. Par conséquent, l'instruction du bloc catch est exécutée. En outre, le code du dernier bloc est également exécuté.

Le bloc final est toujours exécuté, que le bloc try ou le bloc catch soit exécuté ou non.

Le bloc catch nous permet de spécifier le type d'erreur qu'il doit détecter. Dans ce cas, nous essayons de détecter une erreur générale. Par conséquent, nous écrivons

**catch (Exception e)**

où **Exception** fait référence à la classe à laquelle l'erreur appartient et **e** est le nom donné à l'erreur.

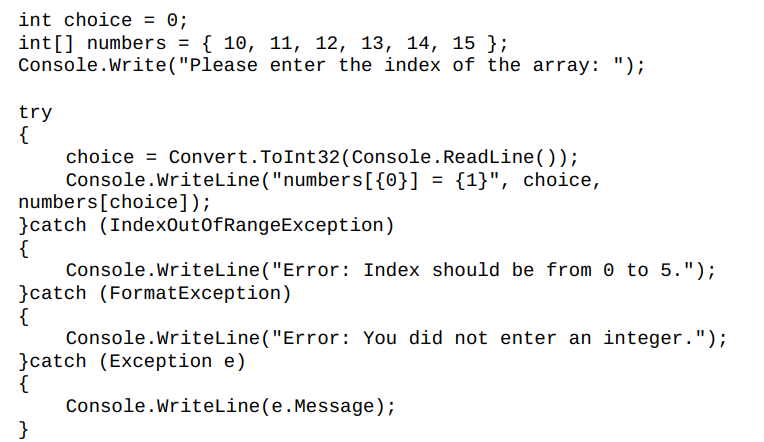
La classe Exception traite toutes les erreurs générales et possède une propriété appelée Message qui explique la raison de l'exception. Pour afficher cette propriété, nous écrivons

Console.WriteLine(e.Message) ;

### **Erreurs spécifiques**

En plus de la classe **Exception** qui traite les erreurs générales, nous avons également d'autres classes qui peuvent traiter des erreurs plus spécifiques. Ceci est utile si vous souhaitez effectuer des tâches spécifiques en fonction de l'erreur détectée. Par exemple, vous pouvez vouloir afficher votre propre message d'erreur.

Essayez d'exécuter le code ci-dessous :



Si vous entrez

10

Vous obtiendrez

L'indice était en dehors des limites du tableau.

Erreur : L'indice doit être compris entre 0 et 5.

Si vous entrez

Bonjour

Vous obtiendrez

La chaîne saisie n'était pas dans un format correct.

Erreur : Vous n'avez pas entré un nombre entier.

La première erreur est une exception IndexOutOfRangeException et a été traitée par le premier bloc catch. Cette exception se produit lorsque vous essayez d'accéder à un élément d'un tableau avec un indice qui est en dehors de ses limites.

La deuxième erreur est une exception FormatException et a été traitée par le deuxième bloc catch. L'exception FormatException se produit lorsque le format d'un argument n'est pas valide. Dans notre exemple, Convert.ToInt32("Hello") a généré une exception FormatException car l'argument "Hello" ne peut pas être converti en entier. En revanche, si vous avez saisi 4, Convert.ToInt32("4") ne générera pas d'exception FormatException car la chaîne "4" peut être convertie en entier.

Après les deux blocs spécifiques catch, nous avons un bloc catch supplémentaire pour détecter les erreurs générales que nous n'avons pas anticipées.

L'exemple ci-dessus montre deux des nombreuses exceptions en C#. Pour une liste complète de toutes les exceptions, consultez le site <https://msdn.microsoft.com/enus/library/system.systemexception.aspx>

# **Chapitre 7 : Programmation Orientée Objet - Partie 1**

Nous avons couvert pas mal de choses jusqu'à présent. Dans les deux prochains chapitres, nous allons nous pencher sur un autre concept important de la programmation - le concept de programmation orientée objet.

Dans ce chapitre, nous apprendrons ce qu'est la programmation orientée objet et comment écrire nos propres classes et créer des objets à partir de celles-ci. En outre, nous aborderons également le concept de champs, de propriétés, de constructeurs et de méthodes.

## **Qu'est-ce que la programmation orientée objet ?**

En termes simples, la programmation orientée objet est une approche de la programmation qui décompose un problème de programmation en objets qui interagissent les uns avec les autres.

Les objets sont créés à partir de modèles connus sous le nom de classes. Vous pouvez considérer une classe comme le plan d'un bâtiment. Un objet est le "bâtiment" réel que nous construisons sur la base du plan.

### **Écrire notre propre classe**

Pour écrire notre propre classe, nous utilisons le mot-clé **class**, suivi du nom de la classe.

Par exemple, pour créer une classe Personnel, nous écrivons

**class Staff {**

**//Contenu de la classe**

**//y compris les champs, les propriétés et les méthodes**

**}**

Il est courant d'utiliser le PascalCasing pour nommer nos classes.

Le PascalCasing désigne la pratique consistant à mettre en majuscule la première lettre de chaque mot, y compris le premier mot (par exemple ThisIsAClassName). C'est la convention que nous suivrons dans le livre.

Le contenu de la classe est inclus dans la paire d'accolades qui suivent le nom de la classe. Le contenu d'une classe comprend les constructeurs, les destructeurs, les constantes, les champs, les méthodes, les propriétés, les indexeurs, les opérateurs, les événements, les délégués, les interfaces, les structures et autres classes.

Dans ce chapitre, nous aborderons certains des éléments les plus courants d'une classe, à savoir les champs, les méthodes, les propriétés et les constructeurs.

Pour comprendre ce que sont ces éléments, nous allons construire une classe à partir de zéro.

Tout d'abord, créez un nouveau projet d'application de console dans Visual Studio Community et nommez ce projet "ClassDemo".

Étudiez le code qui est généré automatiquement pour vous. Remarquez que dans l'espace de noms ClassDemo, VSC a déjà créé une classe appelée Program pour vous ?

Dans la classe Program, nous avons la méthode Main().

Par défaut, la méthode Main() (qui est le point de départ de toutes les applications C#) est placée dans la classe Program créée par VSC. Si nous le voulons, nous pouvons changer le nom de la classe Program pour autre chose, mais la méthode Main() doit être appelé Main(). La méthode Main() doit être présente dans tous les programmes C#.

Dans ce chapitre, nous allons ajouter une deuxième classe à l'espace de noms ClassDemo. Nous appellerons cette nouvelle classe Staff et ajouterons des champs, des propriétés et des méthodes à la classe. Le code complet de ce chapitre peut être téléchargé à l'adresse <http://www.learncodingfast.com/csharp> .

Commençons par déclarer la classe. Ajoutez le code suivant juste avant la ligne class Program dans notre code généré automatiquement.

class Staff

{

}

Nous avons maintenant deux classes dans notre projet : Staff et program.

#### **Champs(Attributs)**

À l'intérieur de la classe Staff, ajoutez les lignes suivantes :

private string nameOfStaff ;

private const int hourlyRate = 30;

private int hWorked;

Ici, nous déclarons une variable string (nameOfStaff) et deux variables int (hourlyRate et hWorked). Ces variables sont connues sous le nom de champs de la classe. Un champ (ou Attribut) est simplement une variable qui est déclarée à l'intérieur d'une classe. Comme toutes les autres variables, elles sont utilisées pour stocker des données.

Vous remarquez qu'il y a un mot "private" devant chaque déclaration ? C'est ce qu'on appelle un modificateur d'accès. Les modificateurs d'accès sont comme des gardiens, ils contrôlent qui a accès à ce champ (c'est-à-dire qui peut lire et modifier la valeur de ce champ).

Un champ peut être soit privé (private), soit public, soit protégé (protected), soit interne (internal). Dans notre cas, nous avons déclaré les trois champs comme étant privés. Cela signifie qu'ils ne sont accessibles qu'à l'intérieur de la classe Staff elle-même.

Il y a deux raisons pour lesquelles nous ne voulons pas que les trois champs soient accessibles en dehors de la classe.

La première raison est qu'il n'est pas nécessaire que les autres classes connaissent ces domaines. Dans notre cas, le champ hourlyRate (tauxHoraire) n'est nécessaire qu'à l'intérieur de la classe Staff. Nous avons une méthode dans la classe Staff qui utilise le champ hourlyRate pour calculer le salaire mensuel d'un employé. Les autres classes n'utilisent pas du tout le champ hourlyRate. Il convient donc de déclarer le champ hourlyRate comme privé afin de le cacher aux autres classes.

C'est ce qu'on appelle l'encapsulation.

#### **L’encapsulation**

L'encapsulation permet à un objet de cacher des données et des comportements à d'autres classes qui n'ont pas besoin de les connaître. Cela nous permet d'apporter plus facilement des modifications à notre code à l'avenir si nécessaire. Nous pouvons modifier en toute sécurité la valeur de hourlyRate dans la classe Staff sans affecter les autres classes.

La deuxième raison de déclarer un champ comme privé est que nous ne voulons pas que les autres classes le modifient librement. Cela permet d'éviter que les champs ne soient corrompus.

Nous parlerons plus en détail des modificateurs d'accès dans le prochain chapitre.

En plus du mot-clé private, nous avons également ajouté le mot-clé const lorsque nous avons déclaré le champ hourlyRate.

private const int hourlyRate = 30 ;

Le mot-clé const indique que la valeur ne peut pas être modifiée après sa création.

Toute variable qui est déclarée comme const doit être initialisée au moment de la déclaration. Dans notre exemple, nous avons initialisé hourlyRate à 30. Cette valeur ne peut être modifiée par la suite nulle part dans le code.

#### **Propriétés**

Ensuite, examinons les propriétés.

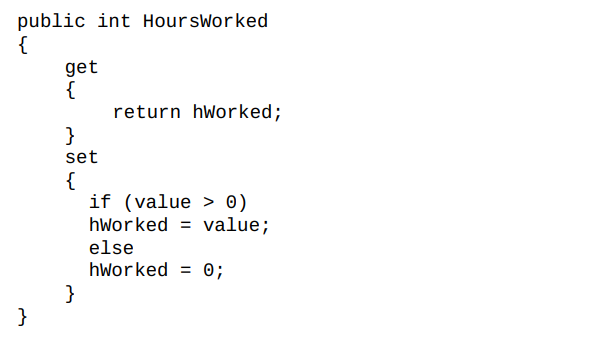
Une propriété est généralement utilisée pour donner accès à un champ privé dans les cas où le champ est nécessaire à d'autres classes. Cela peut sembler contradictoire. Nous avons déjà mentionné que nous utilisons des champs privés pour que les autres classes n'y aient pas accès. Si c'est le cas, pourquoi autorisons-nous l'accès à ces champs via des propriétés ?

L'une des principales raisons est que l'utilisation de propriétés nous permet de mieux contrôler les droits des autres classes lors de l'évaluation de ces champs privés. Nous verrons comment faire cela plus tard.

Pour l'instant, apprenons d'abord comment déclarer une propriété.

Ajoutez les lignes de code suivantes à notre classe Staff, juste après la ligne

private int hWorked ;



Nous avons déclaré notre propriété comme

public int HoursWorked

{

}

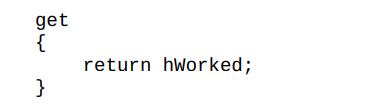
Le modificateur d'accès est public car nous voulons que d'autres classes aient accès à cette propriété.

Le type de données est int car cette propriété est utilisée pour donner accès au champ int privé hWorked. hWorked est connu comme le champ de soutien de la propriété.

Le nom de la propriété est HoursWorked. Nous utilisons normalement le PascalCasing pour les noms de propriété.

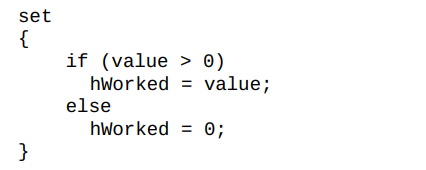
Une propriété contient deux méthodes spéciales connues sous le nom d'accesseurs. Le premier accesseur est un getter et le second est un setter.

Le getter de base renvoie simplement la valeur du champ privé. Ainsi, nous écrivons



où **return** est un mot-clé et hWorked est le nom du champ d'appui.

Le setter fixe la valeur du champ privé. On écrit



value est un mot-clé lorsqu'il est utilisé à l'intérieur d'un setter. Il fait référence à la valeur qui se trouve à droite de la déclaration d'affectation lorsque les utilisateurs utilisent la propriété pour définir la valeur du champ privé. Nous apprendrons comment faire plus tard.

Dans le setter, nous avons fait une vérification simple en utilisant une déclaration if. Nous avons vérifié si la valeur est supérieure à zéro. Si elle l'est, nous l'assignons à hWorked. Sinon, nous mettons hWorked à zéro. Ce setter montre comment nous pouvons utiliser les propriétés pour contrôler les valeurs qui peuvent être attribuées à notre champ privé.

Par défaut, getter et setter ont le même niveau d'accès que la propriété elle-même (publique dans ce cas). Nous n'avons donc pas besoin de les spécifier. Toutefois, si vous ne souhaitez pas que le setter ait le même niveau d'accès que la propriété, vous pouvez déclarer le setter comme privé afin que les autres classes ne puissent pas modifier votre champ privé :

private set

{

}

La propriété est alors une propriété en lecture seule en dehors de la classe Staff. Sa valeur ne peut être fixée qu'à l'intérieur de la classe Staff elle-même.

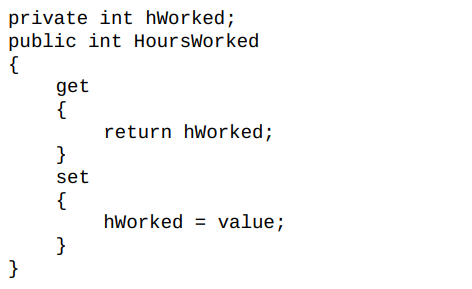
#### **Propriétés auto-implémentées**

Notez que dans les cas où aucune logique supplémentaire n'est requise dans le getter et le setter, C# nous fournit une sténographie pour déclarer la propriété. C'est ce que l'on appelle une propriété auto-implémentée.

Pour déclarer une propriété auto-implémentée, nous écrivons



Cela équivaut à :



Lorsque vous utilisez cette sténographie, vous ne devez pas déclarer un champ privé. Le compilateur créera automatiquement un champ privé anonyme pour vous.

Si vous souhaitez que la propriété soit en lecture seule, vous pouvez la déclarer privée comme ceci :



#### **Méthodes**

Ensuite, examinons les méthodes.

Une méthode est un bloc de code qui exécute une certaine tâche.

Ajoutons une méthode simple à notre classe Staff.

public void PrintMessage ()

{

Console.WriteLine("Calcul de la rémunération...") ;

}

La méthode est déclarée comme :

public void PrintMessage ()

{

}

La déclaration de la méthode indique d'abord le niveau d'accessibilité de la méthode. Ici, nous avons déclaré la méthode comme publique afin que la méthode soit accessible partout dans le programme (pas seulement à l’intérieur la classe Staff).

Ensuite, nous indiquons le type de retour de la méthode. Une méthode peut renvoyer un certain résultat après avoir effectué sa tâche. Si la méthode ne renvoie aucun résultat, nous utilisons le mot-clé **void** comme dans notre exemple.

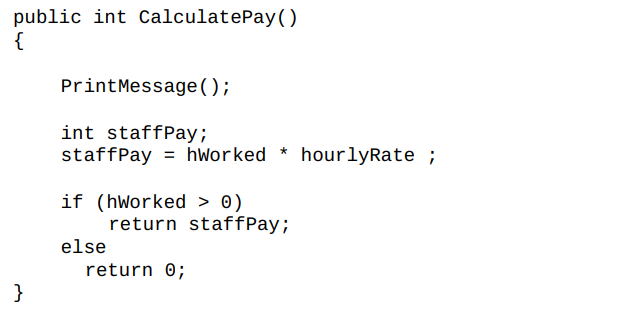
Enfin, nous indiquons le nom de la méthode (PrintMessage dans notre exemple).

La parenthèse () après le nom de la méthode est l'endroit où nous incluons les paramètres de la méthode. Les paramètres sont les noms donnés aux données que nous transmettons à la méthode pour qu'elle puisse accomplir sa tâche. Si la méthode ne nécessite aucune donnée (comme dans notre exemple), nous ajoutons simplement une paire de parenthèses vides après le nom de la méthode.

Après avoir déclaré la méthode, nous définissons ce qu'elle fait à l'intérieur de la paire de parenthèses vides qui suit. C'est ce qu'on appelle l'implémentation de la méthode. Dans notre exemple, la méthode PrintMessage() affiche simplement la ligne "Calcul de la rémunération...".

C'est tout ce que la méthode PrintMessage() permet de faire.

Passons maintenant à une méthode plus complexe. Cette deuxième méthode calcule la rémunération de chaque employé et renvoie le résultat du calcul. Ajoutez les lignes de code suivantes à la classe Staff.



Cette méthode est déclarée comme :

Public int CalculatePay()

{

}

Le mot-clé int indique que cette méthode renvoie une valeur de type int.

À l'intérieur des accolades, nous avons l'énoncé

PrintMessage() ;

C'est ce qu'on appelle la méthode PrintMessage(). Lorsque le programme atteint cet état, il exécute d'abord la méthode PrintMessage() et affiche la ligne "Calculer la rémunération..." avant d'exécuter le reste de la méthode CalculatePay(). Cet exemple montre comment vous pouvez appeler une méthode à l'intérieur d'une autre méthode.

Ensuite, nous déclarons une variable locale appelée staffPay et nous lui attribuons le produit des champs privés hourlyRate et hWorked.

Une méthode peut accéder à tous les champs et propriétés qui sont déclarés à l'intérieur de la classe. En outre, elle peut déclarer ses propres variables. Celles-ci sont connues sous le nom de variables locales et n'existent que dans la méthode. Un exemple est la variable staffPay dans notre exemple.

Après avoir attribué la variable staffPay, nous utilisons une déclaration if pour déterminer le résultat que la méthode doit renvoyer.

Une méthode comporte généralement au moins une déclaration return. return est un mot-clé qui est utilisé pour renvoyer une réponse de la méthode. Il peut y avoir plus d'une déclaration de retour dans une méthode. Cependant, une fois que la méthode exécute une déclaration de retour, la méthode sortira.

Dans notre exemple, si hWorked est supérieur à zéro, le programme exécutera l'instruction

return staffPay ;

et sortir de la méthode. Cette valeur de retour peut ensuite être attribuée à une variable. Par exemple, si hWorked est 10 et hourlyRate est 20, nous pouvons utiliser l'instruction

int pay = CalculatePay() ;

pour affecter le résultat de CalculatePay() à la variable pay. La valeur de la rémunération sera alors de 200.

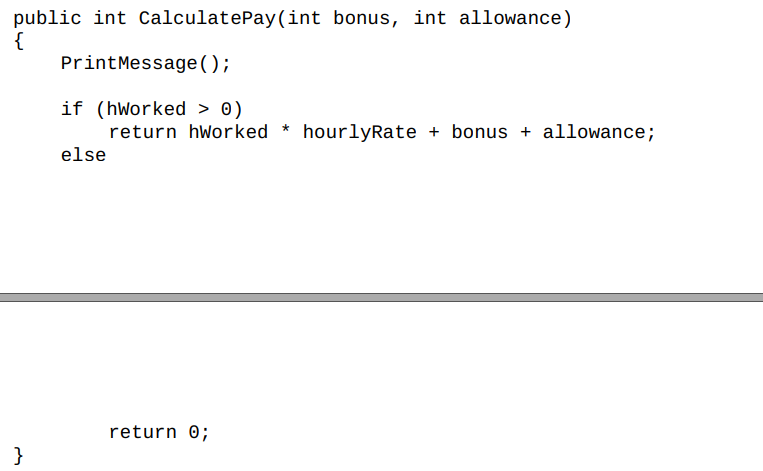
En revanche, si hWorked est inférieur ou égal à zéro, le programme exécutera l'instruction renvoyant 0 ;

et quittera la méthode. La valeur de la rémunération sera 0.

#### **Surcharge**

En C# (et dans la plupart des autres langages), vous pouvez créer deux méthodes du même nom à condition qu'elles aient des signatures différentes. C'est ce qu'on appelle la surcharge. La signature d'une méthode fait référence au nom de la méthode et aux paramètres qu'elle possède.

Ajoutez la méthode suivante en dessous de la méthode précédente CalculatePay().



La signature de la première méthode est CalculatePay() tandis que celle de la seconde méthode est CalculatePay(int bonus, int allowance).

Cette deuxième méthode a deux paramètres - bonus et allocation(indemnité). Elle calcule la rémunération des employés en additionnant les valeurs de ces deux paramètres au produit de hWorked et de hourlyRate. Dans cet exemple, nous n'avons pas utilisé de variable locale pour stocker le résultat de hWorked \* hourlyRate + bonus + allocation.

Nous renvoyons simplement le résultat du calcul directement. C'est tout à fait correct.

Nous apprendrons plus tard comment utiliser cette méthode.

#### **La méthode ToString()**

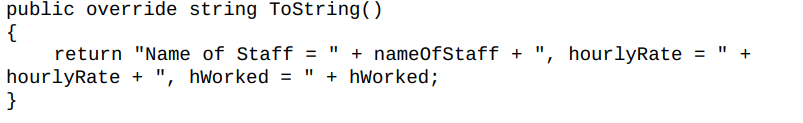
Enfin, avant de passer à la section suivante, nous devons écrire une autre méthode - la méthode ToString().

La méthode ToString() est une méthode spéciale qui renvoie une chaîne de caractères représentant la classe courante. En C#, toutes les classes sont livrées avec une méthode ToString() prédéfinie.

Cependant, il est d'usage (et attendu de nous) de surcharger cette méthode.

Surcharger une méthode signifie simplement écrire notre propre version de la méthode.

En général, la méthode ToString() que nous écrivons affiche les valeurs des champs et des propriétés de la classe. Ajoutez le code suivant à la classe Staff :



Comme vous pouvez le voir, la méthode **ToString**() renvoie un type de chaîne de caractères. La chaîne qu'elle renvoie contient des informations sur la classe Staff. Le mot-clé **override** dans la déclaration de la méthode indique que cette méthode remplace la méthode par défaut.

Nous aborderons plus en détail le mot-clé **override** dans le prochain chapitre.

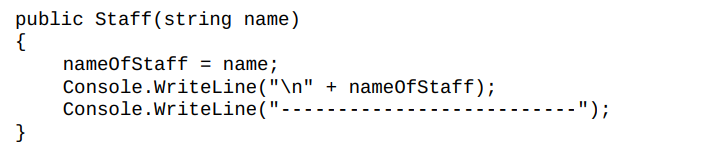
#### **Constructeurs**

Maintenant, regardons les constructeurs.

Un constructeur est une méthode spéciale qui est utilisée pour "construire" un objet à partir du modèle de classe. C'est la première méthode qui est appelée chaque fois que nous créons un objet à partir de notre classe. Les constructeurs sont couramment utilisés pour initialiser les champs de la classe.

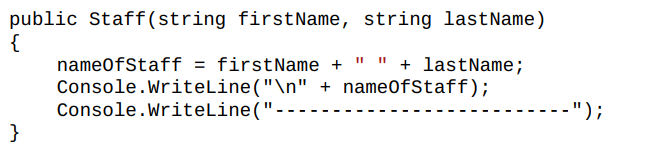
Un constructeur a toujours le même nom que la classe (Staff dans notre cas) et ne renvoie aucune valeur. Nous n'avons pas besoin d'utiliser le mot-clé void lorsque nous déclarons un constructeur.

Ajoutez les lignes suivantes à notre classe Staff.



Dans ce constructeur, nous initialisons d'abord le champ nameOfStaff avec la chaîne de caractères qui est passée au constructeur (name). Nous affichons ensuite la valeur de nameOfStaff à l'écran et la soulignons par une série de tirets.

Comme toute autre méthode, nous pouvons avoir plus d'un constructeur tant que la signature est différente. Nous pouvons ajouter un autre constructeur à notre classe.



Ce constructeur a deux paramètres - prénom et nom. La première ligne concatène les deux chaînes et attribue la chaîne résultante à nameOfStaff.

Les deux lignes suivantes affichent le nom du personnel à l'écran et le soulignent par une série de tirets.

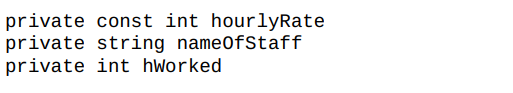
La déclaration d'un constructeur est facultative. Si vous ne déclarez pas votre propre constructeur, C# en crée un automatiquement pour vous. Le constructeur par défaut initialise simplement tous les champs de la classe à des valeurs par défaut, qui sont normalement zéro pour les champs numériques et une chaîne vide pour les champs de type chaîne.

#### **L'instanciation d'un objet**

Maintenant que nous savons comment créer une classe, voyons comment nous pouvons utiliser la classe pour créer un objet. Ce processus est connu sous le nom d'instanciation d'un objet. Un objet est également connu sous le nom d'instance.

Pour résumer, notre classe Staff comporte les éléments suivants :

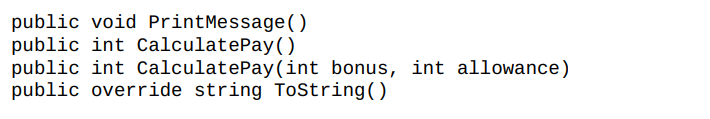
**Champs ou Attributs**



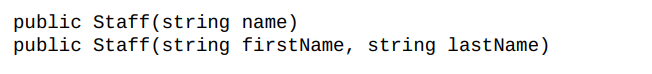
**Propriétés**



**Méthodes**



**Constructeurs**

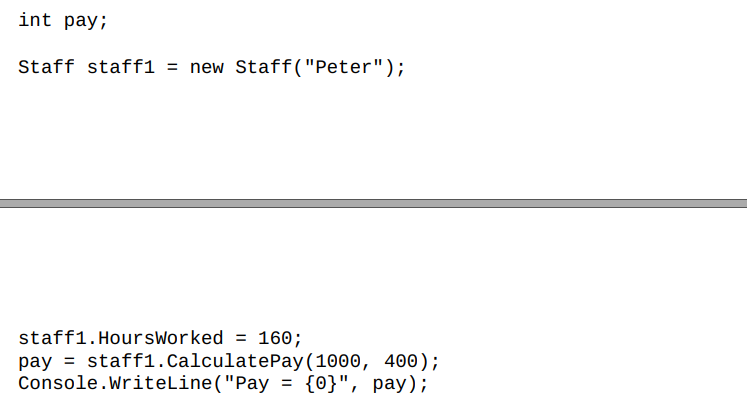


Nous allons instancier un objet Staff dans la méthode Main() à l'intérieur de la classe Program.

La syntaxe pour l'instanciation d'un objet est

**ClassName objectName = new ClassName(arguments) ;**

Ajoutez les lignes suivantes à l'intérieur des accolades de la méthode Main() dans la classe Program.



Ici, nous utilisons le premier constructeur (avec un paramètre) pour créer notre objet staff1.

Une fois l'objet créé, nous pouvons utiliser l'opérateur point après le nom de l'objet pour accéder à tout champ, propriété ou méthode publique dans la classe Staff. Notez que nous devons utiliser l'opérateur de point ici car nous essayons d'accéder aux membres de la classe Staff de la classe Program. L'opérateur de point est nécessaire chaque fois que l'on veut accéder à un champ, une propriété ou une méthode d'une autre classe.

Si vous accédez à des membres de la même classe, vous n'avez pas besoin d'utiliser l'opérateur de point. Un exemple est lorsque nous avons appelé la méthode PrintMessage() de la méthode CalculatePay() plus tôt. Nous n'avons pas utilisé l'opérateur de point car les deux méthodes appartiennent à la même classe (Staff).

Après avoir créé notre objet staff1, la ligne suivante montre comment nous pouvons utiliser la propriété publique EmployeeType pour attribuer une valeur au champ hWorked.

staff1.HoursWorked = 160 ;

Si nous essayons d'accéder au champ privé hWorked directement en écrivant

staff1.HoursWorked = 160 ;

nous obtiendrons une erreur car hWorked est un domaine privé et n'est donc accessible que dans la classe Staff.

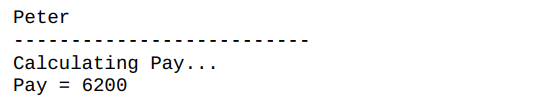
Pour appeler la méthode CalculatePay(), nous écrivons

staff1.CalculatePay(1000, 400) ;

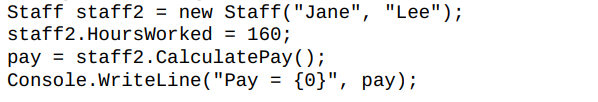
Dans cet exemple, comme nous avons les nombres 1000 et 400 entre parenthèses, nous utilisons la deuxième méthode CalculatePay(). Nous passons les valeurs 1000 et 400 aux paramètres prime et allocation respectivement. Les valeurs que nous avons passées sont connus sous le nom d'arguments. Le programme utilise ensuite cette méthode pour calculer le salaire et renvoyer la réponse. Cette réponse est affectée à la variable pay.

Enfin, nous utilisons la méthode Console.WriteLine() pour afficher la valeur de la rémunération à l'écran.

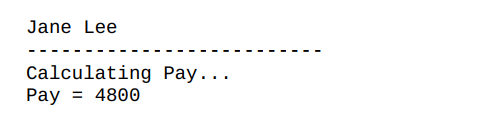
Si vous exécutez le code ci-dessus, vous obtiendrez :



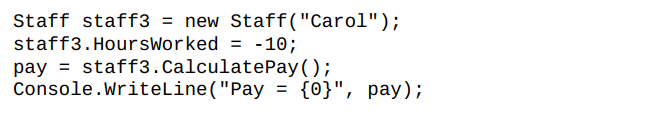
Jouez un peu avec le code pour mieux comprendre le fonctionnement des cours. Essayez d'ajouter les lignes de code suivantes



Si vous exécutez le code ci-dessus, vous obtiendrez

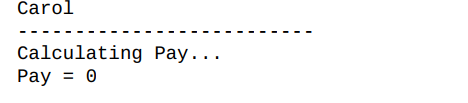


Enfin, créons un troisième objet pour démontrer comment fonctionne la validation des données lorsque nous utilisons des propriétés. Ajoutez les lignes de code suivantes.



Ici, nous avons essayé de fixer la propriété des heures travaillées à -10, ce qui est une valeur non valable.

L'auteur de cette propriété fixe plutôt la valeur à zéro. Si vous exécutez ce code, vous obtiendrez



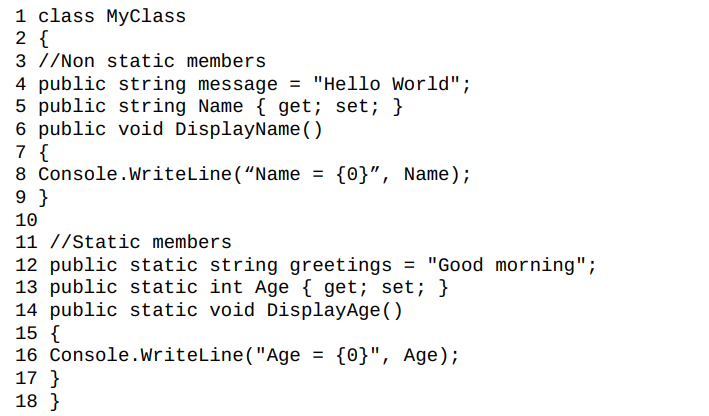
#### **Mot-clé static**

Nous avons abordé des concepts assez compliqués dans ce chapitre. Je vous suggère vivement de télécharger le programme complet de ce chapitre à l'adresse <http://www.learncodingfast.com/csharp> et de vous amuser avec. Étudiez le code et assurez-vous de bien comprendre les sujets abordés dans ce chapitre avant de passer à autre chose.

Dans cette section, nous examinerons un autre mot-clé qui est parfois utilisé lorsque nous déclarons des classes ou des membres de classe (c'est-à-dire des méthodes, des champs, des propriétés, des constructeurs, etc.)

Auparavant, nous avons examiné comment nous pouvons utiliser la classe Staff pour créer nos objets staff1, staff2 et staff3. Toutefois, certaines classes ou certains membres de classe sont accessibles sans qu'il soit nécessaire de créer des objets. Il s'agit des classes ou membres de classe statiques, qui sont déclarés à l'aide du mot-clé static.

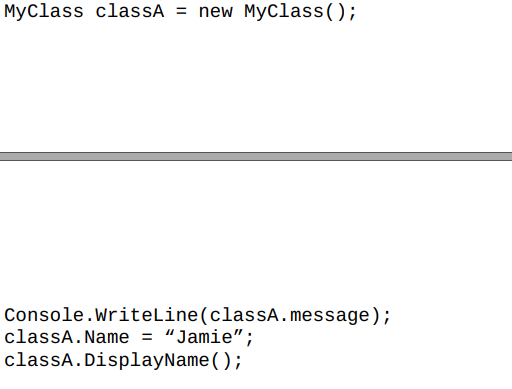
Prenons la classe suivante :



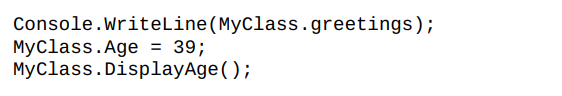
MyClass contient un champ non statique message, une propriété non statique Name et une méthode non statique DisplayName() (lignes 4 à 9).

Elle contient également un champ statique greetings, une propriété statique Age et une méthode statique DisplayAge() (lignes 12 à 17).

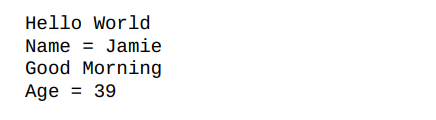
Pour accéder aux **membres non statiques de MyClass depuis une autre classe,** il faut instancier un objet comme précédemment :

****

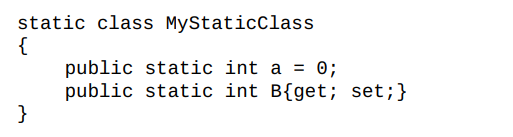
Cependant, **pour accéder aux membres statiques**, il n'est pas nécessaire de créer un objet. Nous utilisons simplement le nom de la classe pour y accéder comme indiqué ci-dessous.



Si vous exécutez le code ci-dessus, vous obtiendrez le résultat suivant :



En plus d'avoir des méthodes, des champs, des propriétés et des constructeurs statiques, nous pouvons aussi avoir des classes statiques. Une classe statique ne peut contenir que des membres statiques. Un exemple est présenté ci-dessous.

****

Certaines des classes pré-écrites en C# sont déclarées comme des classes statiques. Un exemple est la classe Console. Nous n'avons pas besoin de créer un objet Console lorsque nous utilisons des méthodes de la classe Console. Nous écrivons simplement Console.WriteLine("Bonjour le monde") ;.

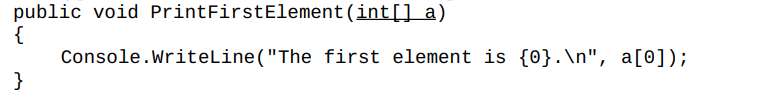
### **Concepts de méthodes avancées**

Maintenant que vous êtes familiarisé avec les classes, passons à quelques concepts avancés concernant la déclaration et l'utilisation des méthodes dans une classe. Ces concepts sont plus complexes et peuvent nécessiter plus d'une lecture pour les comprendre pleinement.

#### **Utilisation des tableaux et des listes**

Auparavant, nous avons appris à utiliser des types de données de base comme int et float comme paramètres d'une méthode. En plus des types de données de base, nous pouvons également utiliser des tableaux et des listes.

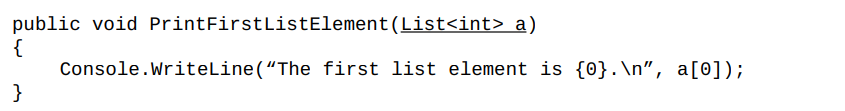
Pour utiliser un tableau comme paramètre, nous ajoutons un crochet [] après le type de données du paramètre dans la déclaration de la méthode. Un exemple est présenté ci-dessous.

****

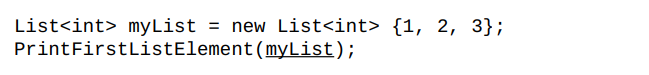
Pour appeler cette méthode, nous devons déclarer un tableau et le transmettre comme argument à la méthode :

****

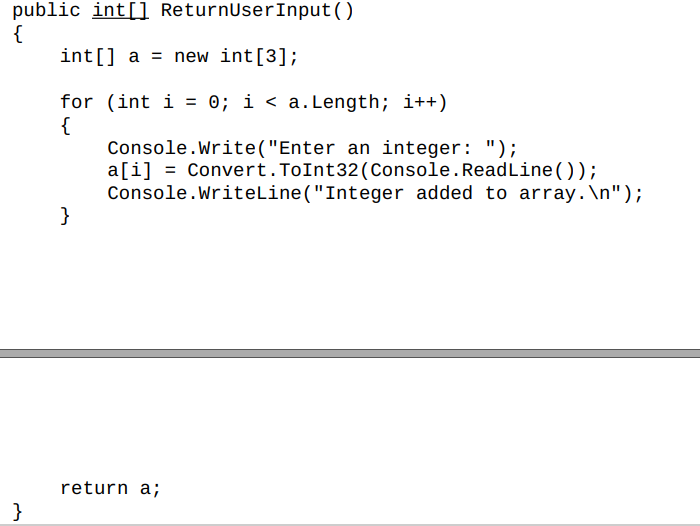
L'exemple suivant montre comment nous pouvons utiliser une liste comme paramètre.



Pour appeler la méthode, nous devons déclarer une liste et la transmettre comme argument à la méthode.



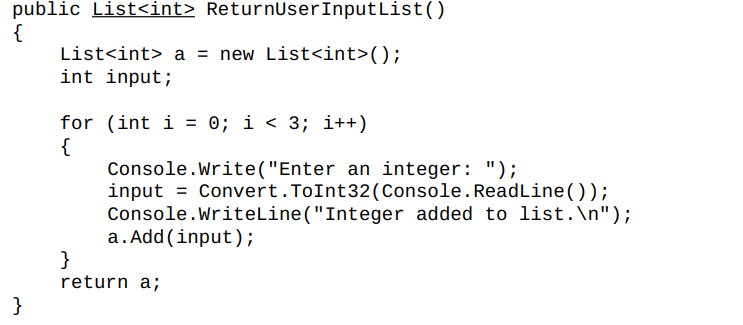
En plus d'utiliser des tableaux ou des listes comme paramètres d'une méthode, nous pouvons également retourner un tableau ou une liste d'une méthode. Pour retourner un tableau à partir d'une méthode, nous ajoutons un crochet [] après le type de retour dans la déclaration de la méthode.

****

Pour utiliser cette méthode, nous devons déclarer un tableau et lui attribuer le résultat de la méthode.

int[] myArray2 = ReturnUserInput() ;

Pour retourner une liste à partir d'une méthode, nous utilisons le mot-clé List<> comme type de retour dans la déclaration de la méthode. Un exemple est



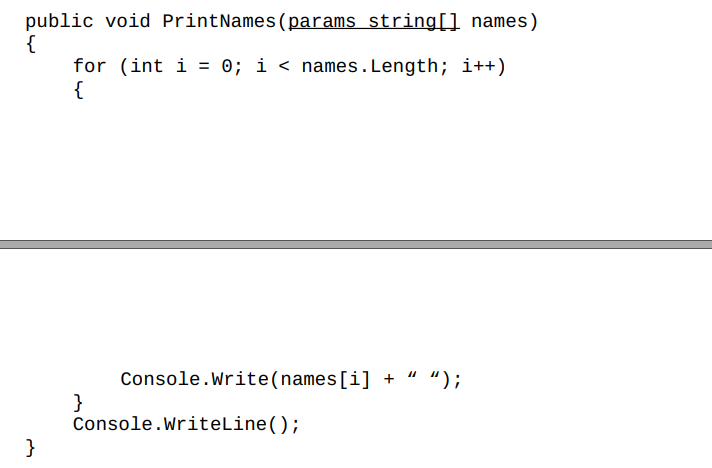
Pour utiliser cette méthode, nous devons déclarer une liste et lui attribuer le résultat de la méthode.

List<int> myList2 = ReturnUserInputList();

#### **Utilisation du mot-clé params**

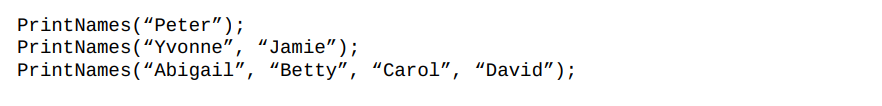
Ensuite, nous allons explorer le mot-clé params. Le mot-clé params est utile lorsque nous ne connaissons pas le nombre d'arguments d'une méthode. Par exemple, nous pouvons avoir une méthode qui affiche une série de noms, mais nous ne savons pas à l'avance combien de noms il y a. Dans de tels cas, nous pouvons utiliser un tableau comme paramètre et ajouter le mot-clé params devant celui-ci.

Un exemple est

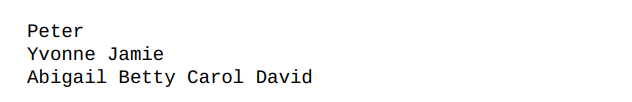


Pour utiliser cette méthode, on peut faire passer un nombre quelconque de chaînes de caractères comme arguments.

Exemple

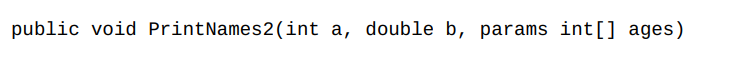


Sortie

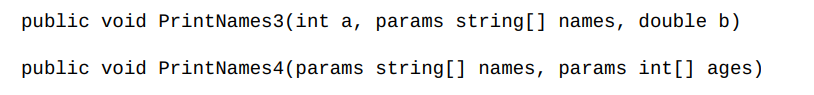


Notez qu'aucun paramètre supplémentaire n'est autorisé après le mot-clé params dans une déclaration de méthode, et qu'un seul mot-clé params est autorisé dans une déclaration de méthode.

Par conséquent, la déclaration de méthode suivante est acceptable



mais les déclarations suivantes ne le sont pas



PrintNames3 n'est pas autorisé parce que le double b vient après les noms de la chaîne params[].

PrintNames4 n'est pas autorisé parce qu'il y a deux mots-clés params.

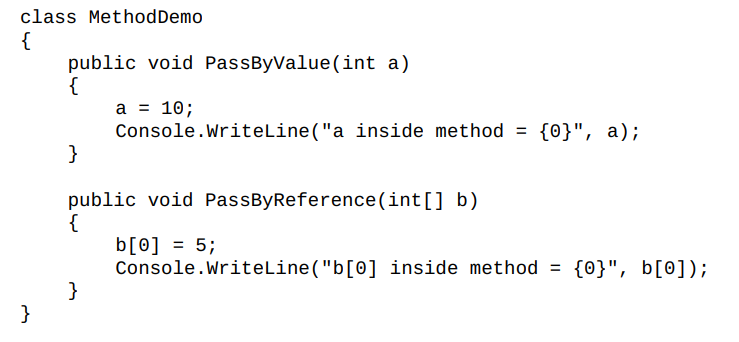
### **Paramètres de Type Référence vs Type de Valeur de Passage**

J'espère que vous avez maintenant une bonne compréhension du fonctionnement des classes et des méthodes. Avant de terminer ce chapitre, j'aimerais revenir sur le concept des types de données de valeur et des types de données de référence. Dans le chapitre 4, nous avons appris qu'il existe deux grandes catégories de types de données en C# - les types de valeurs et les types de référence. Il y a une différence lorsque vous passez une variable de type valeur à une méthode par rapport à une variable de type référence.

Lorsque vous transmettez une variable de type valeur, toute modification apportée à la valeur de cette variable n'est valable que dans la méthode elle-même. Une fois que le programme quitte la méthode, la modification n'est plus valable.

En revanche, si vous introduisez une variable de type référence, toute modification apportée à la variable est valable même après la fin de la méthode.

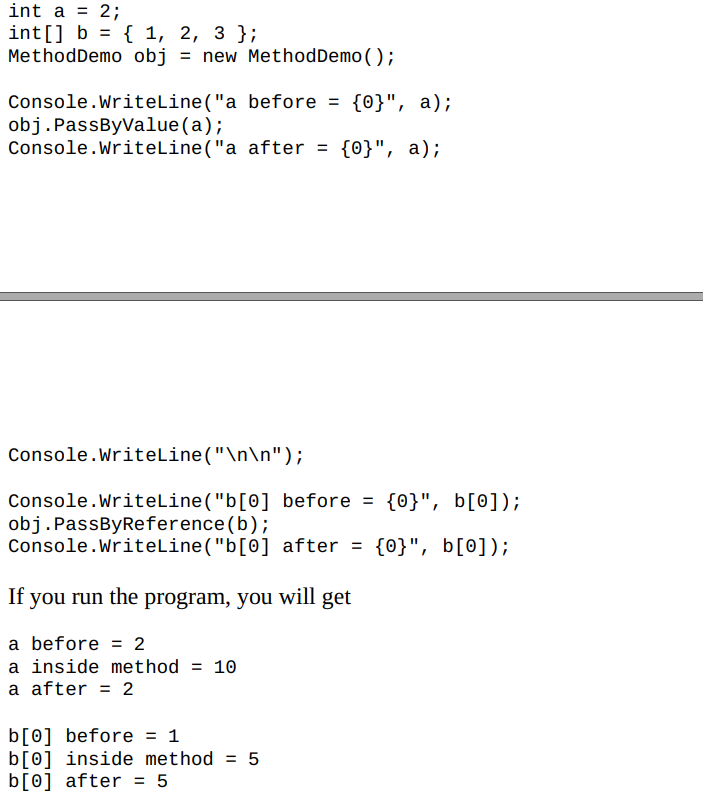
Considérez la classe ci-dessous :



Au sein de la classe, nous avons deux méthodes. La première méthode accepte une variable de type valeur et essaie de changer la valeur de cette variable. Elle affiche ensuite la valeur de la variable.

La deuxième méthode accepte un tableau (type de référence) et essaie de changer la valeur du premier élément du tableau. Elle affiche ensuite la valeur de cet élément.

Dans notre programme Main(), supposons que nous ayons les lignes de code suivantes :



La valeur de a reste la même avant et après l'appel de méthode ; le changement n'est valable qu'à l'intérieur de la méthode elle-même.

En revanche, la valeur de b[0] change après l'appel de la méthode.

Soyez conscient de cette différence lorsque vous passez une variable de type valeur à une méthode (par exemple int, float, etc.) par rapport à une variable de type référence (comme un tableau ou une liste).

# **Chapitre 8 : Programmation axée sur les objectifs - Partie 2**

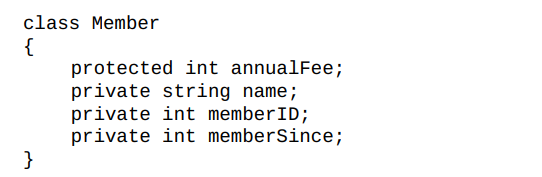
Passons maintenant à certains des sujets les plus avancés en matière de programmation orientée objet. Dans ce chapitre, nous aborderons l'héritage, le polymorphisme, les classes abstraites et les interfaces.

## **Héritage**

L'héritage est l'un des concepts clés de la programmation orientée objet. En termes simples, l'héritage nous permet de créer une nouvelle classe à partir d'une classe existante afin de pouvoir réutiliser efficacement le code existant.

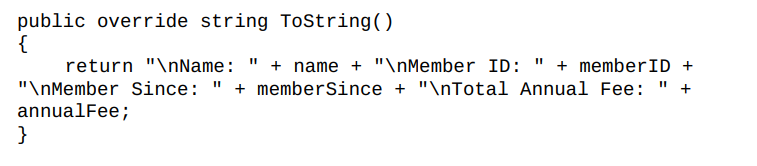
### **Écriture de la classe parent**

Supposons que nous écrivions un programme pour un club de fitness qui a deux types de membres - VIP et Normal. Pour ce faire, créons d'abord une classe appelée Membre.

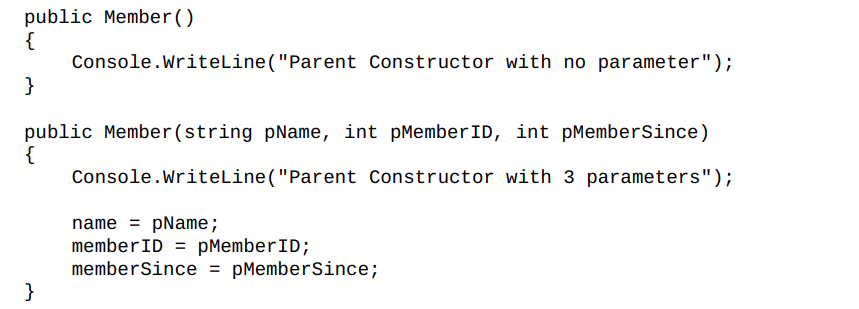
****

Member (la classe Membre) contient un champ protégé et trois champs privés. Un champ protégé est un champ qui n'est accessible que dans la classe dans laquelle il est déclaré et dans toute classe qui en dérive. Nous parlerons très bientôt des classes dérivées.

Ensuite, écrivons une méthode ToString() pour afficher les valeurs des quatre champs.



Enfin, ajoutons deux constructeurs à la classe Member.



Le premier constructeur se contente d’afficher la ligne "Parent constructeur with no parameter".

Le deuxième constructeur est plus intéressant. Il affiche la ligne "Parent Constructeur with 3 parameters" et attribue ses paramètres aux trois champs privés de la classe Membre.

### **Ecriture de la classe enfant**

Maintenant, apprenons comment dériver une classe de la classe Member. Les classes dérivées sont appelées classes enfant, tandis que les classes dont elles sont dérivées sont appelées classes parent ou classes de base.

Une classe dérivée hérite de tous les membres publics et protégés de la classe parent. En d'autres termes, elle peut utiliser ces champs, propriétés et méthodes comme s'ils faisaient partie de son propre code.

Notre classe parent (Member) a le contenu suivant :

**Champs ou attributs**

protected int annualFee

private string name

private int memberID

private int memberSince

**Méthodes**

public override string ToString()

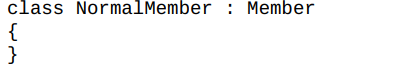
**Constructeurs**

public Member()

public Member(string pName, int pMemberID, int pMemberSince)

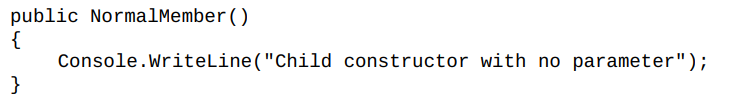
Nous dérivons deux classes - NormalMember et VIPMember - de la classe Member.

D'abord, déclarons la classe enfant NormalMember. Nous indiquons qu'elle est dérivée de la classe Membre en utilisant un deux-points ( :) comme ceci



Maintenant, nous devons écrire le constructeur pour la classe enfant. Le constructeur d'une classe enfant est construit sur le constructeur du parent. Chaque fois que nous créons un objet enfant, le constructeur de la classe parent est toujours appelé en premier.

Il y a deux façons de créer un constructeur enfant. La première consiste à le déclarer simplement comme n'importe quel autre constructeur.

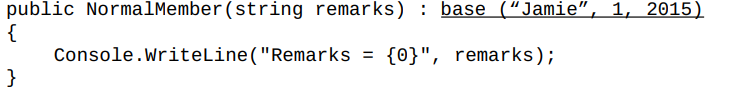


Lorsque nous déclarons notre constructeur comme ci-dessus, C# cherche un constructeur sans paramètre dans la classe parent et l'appelle d'abord avant d'exécuter le code dans le constructeur enfant. Si vous utilisez ce constructeur pour créer un objet enfant, les deux lignes suivantes seront affichées à l'écran :



La première ligne provient du constructeur parent et la deuxième ligne du constructeur enfant.

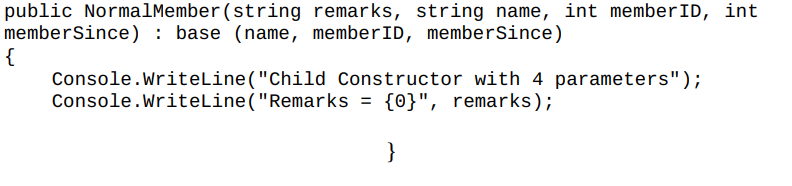
La deuxième façon de déclarer un constructeur enfant consiste à utiliser le signe deux points ( :) et le mot-clé de base pour appeler un constructeur non paramétrique dans la classe parent. Un exemple est présenté ci-dessous :



Lorsque nous appelons un constructeur sans paramètres dans la classe parent, nous devons passer des valeurs à ses paramètres. Dans l'exemple ci-dessus, nous avons passé les valeurs "Jamie", 1 et 2015 au constructeur parent. Ces valeurs sont ensuite attribuées respectivement aux champs name, memberID et memberSince dans la classe de base.

Dans cet exemple, nous avons transmis des valeurs fixes comme arguments au constructeur de base. Cependant, il est préférable de passer les arguments par le constructeur enfant.

L'exemple ci-dessous montre comment cela peut être fait. Remplacez le constructeur précédent par le constructeur ci-dessous :

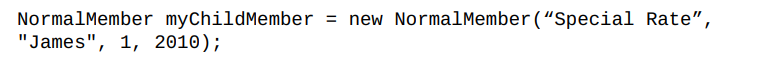


Ce nouveau constructeur enfant a quatre paramètres. Le premier paramètre est un paramètre de chaîne de caractères appelé remarks. Ce paramètre est utilisé à l'intérieur du constructeur enfant.

Les deuxième, troisième et quatrième paramètres ne sont pas utilisés dans le constructeur enfant.

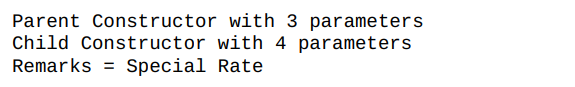
Ils sont plutôt transmis au constructeur parent sous forme d'arguments en fonction de leur nom. Par exemple, le deuxième paramètre du constructeur enfant (string name) est transmis comme premier argument au constructeur parent (name).

Lorsque nous créons un objet enfant avec ce constructeur, nous écrivons quelque chose comme

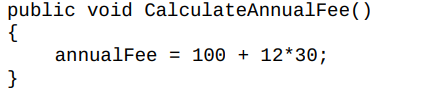


Le constructeur de base avec 3 paramètres est appelé et exécuté en premier. Les valeurs "James", 1 et 2010 sont passées au constructeur de base. Dans les coulisses, ces valeurs sont attribuées respectivement aux champs name, memberID et memberSince dans la classe de base. Après l'exécution du constructeur de base, le constructeur enfant sera exécuté. La chaîne "Special Rate" est attribuée au paramètre remarks et affichée à l'écran.

Lorsque vous exécutez le code, vous obtenez le résultat suivant

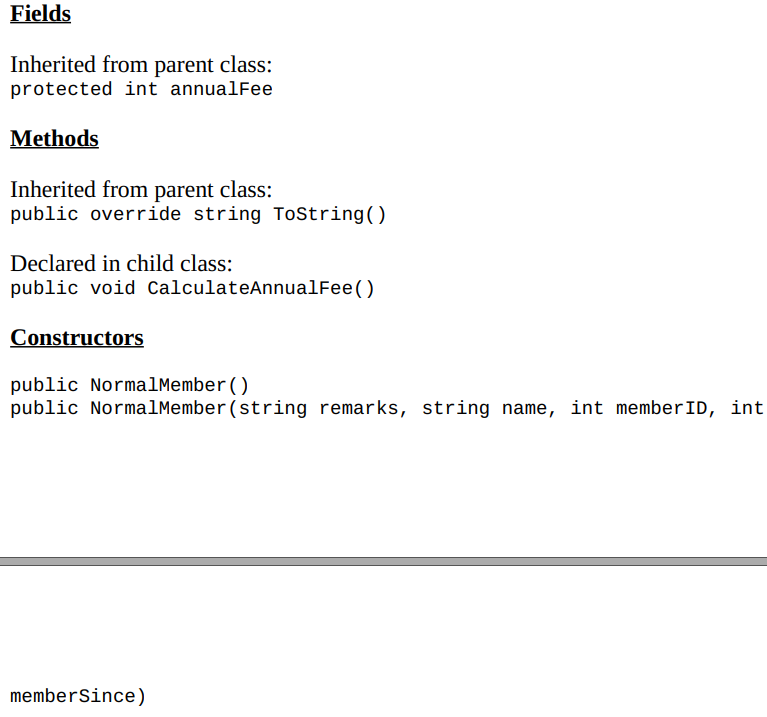
****

Maintenant que nous avons créé les constructeurs pour notre classe enfant, passons à la création d'une méthode pour calculer la cotisation annuelle d'un membre normal. Le code est simplement.

****

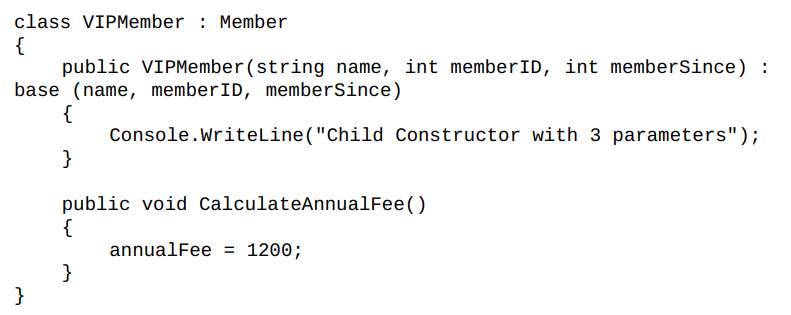
Lorsque nous écrivons "annualFee" dans le code ci-dessus, nous accédons au champ protégé annualFee dans la classe parent. Rappelez-vous qu'une classe enfant a accès à tous les champs publics et protégés de sa classe parent ? Par conséquent, la classe enfant peut utiliser ce champ comme s'il s'agissait de son propre champ. La classe enfant n'a pas besoin de créer une instance de la classe parent pour accéder à ses champs protégés.

C'est tout pour notre classe enfant NormalMember. La classe a le contenu suivant:



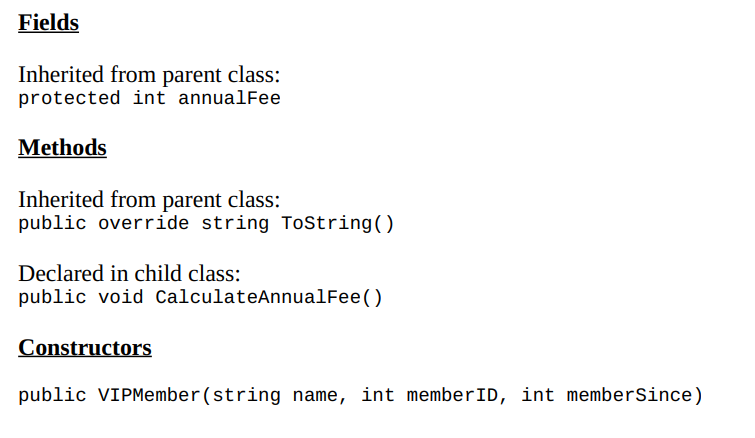
Avc Inherited from parent class = héritage de la classe parent, declared in child class = déclaration dans la classe enfant

Ensuite, écrivons une autre classe qui hérite de Membre. Cette fois, la classe dérivée s'appelle VIPMember. Le code est indiqué ci-dessous.



Cette classe a un constructeur (avec 3 paramètres) et une méthode CalculateAnnualFee(). La méthode CalculateAnnualFee() utilise ici une formule de calcul de la cotisation annuelle différente de celle de la méthode CalculateAnnualFee() de la classe NormalMember. Il est normal que les deux méthodes partagent le même nom (et la même signature) car elles sont dans des classes différentes.

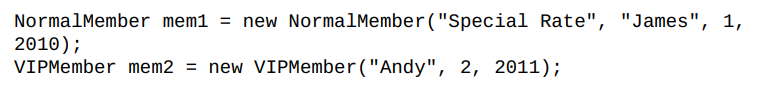
La classe VIPMember a le contenu suivant :



### **La méthode Main()**

Maintenant que nous avons écrit les trois classes dont nous avons besoin, écrivons le code de la méthode Main().

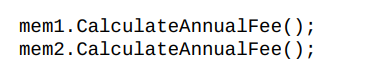
Tout d'abord, nous allons créer deux objets à partir des deux classes dérivées.



mem1 est créé en utilisant le constructeur 4 paramètres de la classe NormalMember.

mem2 est créé en utilisant le constructeur à 3 paramètres de la classe VIPMember.

Ensuite, nous utiliserons les méthodes CalculateAnnualFee() dans les classes respectives.



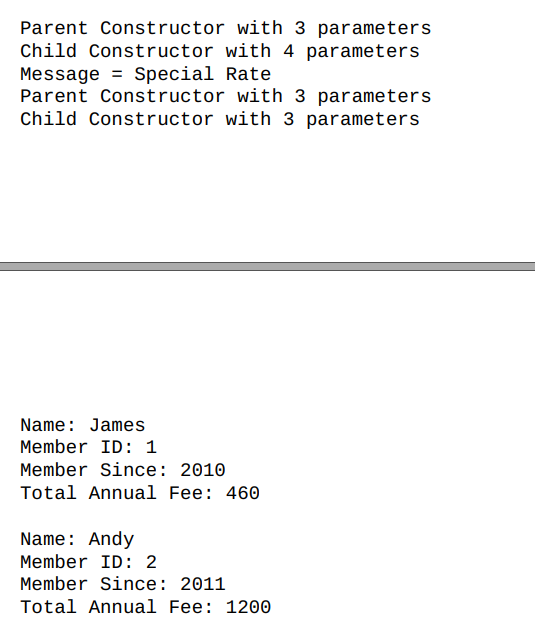
Comme mem1 est une instance de la classe NormalMember, la méthode CalculateAnnualFee() de cette classe est exécutée. La cotisation annuelle pour mem1 est donc de 100 + 12\*30 = 460. Pour mem2, la cotisation annuelle est de 1200 car elle utilise la méthode de la classe VIPMember.

Enfin, utilisons la méthode ToString() de la classe parent (Member) pour afficher les informations sur notre écran. Nous écrivons



Comme la méthode ToString() appartient à la classe parent et est publique, mem1 et mem2 ont tous deux hérité de la méthode et peuvent donc l'utiliser dans la méthode Main(). Cela facilite la réutilisation du code car nous n'avons pas besoin de réécrire la méthode ToString() pour les deux classes enfant.

Vous obtiendrez les résultats suivants lorsque vous exécuterez le programme :

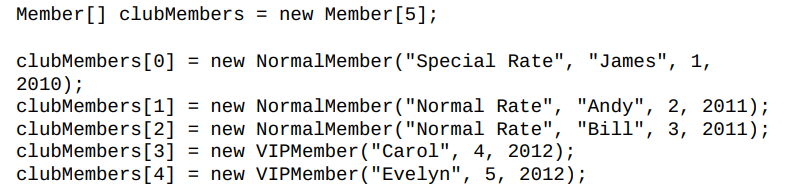


## **Polymorphisme**

Maintenant que nous avons vu un exemple du fonctionnement de l'héritage, passons à un autre sujet étroitement lié à l'héritage : le concept de polymorphisme. Le polymorphisme fait référence à la capacité d'un programme à utiliser la bonne méthode pour un objet en fonction de son type d'exécution.

La meilleure façon d'expliquer le polymorphisme est de prendre un exemple. Développons notre exemple de club de fitness ci-dessus.

Tout d'abord, supprimez tout le code de la méthode Main() précédente et ajoutez les lignes suivantes :

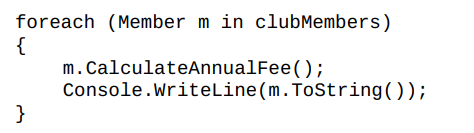


Ici, nous déclarons un tableau de types de membres et nous y ajoutons 5 membres. Les trois premiers membres sont des instances de la classe NormalMember, tandis que les deux derniers sont des instances de la classe VIPMember.

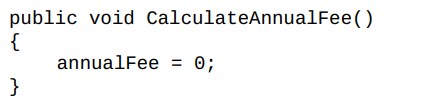
Bien que clubMembers soit déclaré comme un tableau de type de membre, nous pouvons lui attribuer des instances de NormalMember et de VIPMember car ce sont des classes enfant de la classe Member. Nous n'avons pas besoin de déclarer des tableaux séparés pour les objets NormalMember et VIPMember.

Ensuite, nous utiliserons une boucle de foreach pour calculer la cotisation annuelle de chaque membre et afficher les informations.

Pour ce faire, nous écrivons



Si vous essayez d'exécuter le programme à ce stade, vous obtiendrez une erreur indiquant que le terme "Member" ne contient pas de définition pour " CalculateAnnualFee ". En effet, clubMembers est déclaré comme étant un tableau de types Member. Par conséquent, le compilateur essaie d'exécuter la méthode CalculateAnnualFee() dans la classe Member lorsque nous écrivons m.CalculateAnnualFee(). Une erreur se produit parce que nous n'avons pas cette méthode dans notre classe parent Member ; nous ne l'avons que dans les deux classes enfants. Pour corriger cette erreur, nous devons ajouter la méthode suivante à notre classe parent.

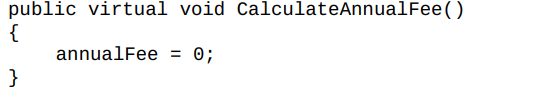


Maintenant, lancez le programme et faites attention au "Total des frais annuels" de chaque membre. Que remarquez-vous ? Tout devrait afficher 0 $. Cela signifie que la méthode CalculateAnnualFee() qui est appelée est celle de la classe mère.

Cela n'est pas surprenant car le clubMembers est déclaré comme étant de type Member.

Si vous souhaitez que la méthode enfant soit appelée à la place, vous devez effectuer deux changements.

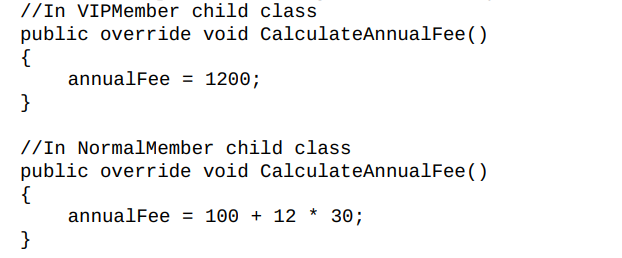
Tout d'abord, vous devez déclarer la méthode parent comme virtuelle, comme ceci



#### **Virtual**

Le mot-clé virtual indique au compilateur que cette méthode peut être remplacée par la méthode des classes dérivées. Lorsque le compilateur rencontre ce mot-clé, il cherche la même méthode dans la classe dérivée et exécute cette méthode à la place.

Ensuite, dans la classe dérivée, vous devez déclarer que votre méthode surpasse la méthode de la classe parent en utilisant le mot-clé override, comme ceci



Maintenant, si vous renouvelez le programme, la cotisation annuelle des trois premiers membres (NormalMember) et des deux derniers membres (VIPMember) sera respectivement de 460 $ et de 1200 $.

C'est le résultat du polymorphisme. Au moment de l'exécution (c'est-à-dire lorsque le programme est lancé), le programme détermine que les trois premiers membres de clubMembers sont de type NormalMember et exécute la méthode CalculateAnnualFee() à partir de cette classe. Il détermine également que les deux derniers membres sont de type VIPMember et exécute la méthode de cette classe.

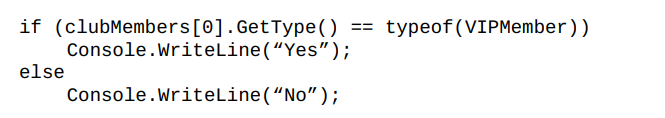
Le polymorphisme signifie simplement qu'au moment de l'exécution, le programme est suffisamment intelligent pour utiliser la méthode CalculateAnnualFee() de la classe enfant correcte même lorsque cet objet est déclaré de type Member.

Nous disons que le type d'exécution des trois premiers éléments de clubMembers est NormalMember tandis que le type d'exécution des deux derniers éléments est VIPMember. Le type déclaré de tous les 5 éléments est Member.

### **GetType() et typeof()**

Dans l'exemple précédent, nous avons laissé le programme déterminer le type de durée d'exécution de chaque membre du tableau clubMembers et appeler la méthode correcte CalculateAnnualFee(). Cependant, il peut parfois être nécessaire pour nous de déterminer nous-mêmes le type de durée d'exécution de chaque membre individuel lorsque nous codons. Nous en verrons un exemple plus tard dans notre projet.

La déclaration if ci-dessous montre comment vous pouvez déterminer si le premier élément du tableau clubMember est de type VIPMember au moment de l'exécution :



#### **GetType()**

La méthode GetType() renvoie le type d'exécution d'un objet.

#### **typeOf()**

La méthode typeof() prend le nom d'un type de données (par exemple int, float, ou le nom d'une classe) et renvoie le type de ce nom, que nous pouvons ensuite comparer avec le résultat de la méthode GetType() sur la gauche.

Si vous exécutez le code ci-dessus, vous obtiendrez "No" comme résultat puisque clubMembers[0] n'est pas de type VIPMember.

## **Classes et méthodes d'abstraction**

Maintenant que nous sommes familiarisés avec l'héritage (et le polymorphisme), passons à l'examen de deux types particuliers de "classe parent" en C# - les classes abstraites et les interfaces.

Examinons d'abord les classes abstraites.

#### **Classe abstraite**

Une classe abstraite est un type spécial de classe qui est créée strictement pour être une classe de base dont les autres classes peuvent dériver. Elles ne peuvent pas être instanciées. En d'autres termes, si FourWheelVehicles est une classe abstraite, l'énoncé

FourWheelVehicle myVeh = new FourWheelVehicle();

vous donnera une erreur car vous ne pouvez pas créer un objet d'une classe abstraite.

Les classes abstraites peuvent avoir des champs, des propriétés et des méthodes comme n'importe quelle autre classe. Cependant, elles ne peuvent pas avoir de membres statiques. En outre, les classes abstraites peuvent avoir un type spécial de méthode appelé méthodes abstraites.

#### **Méthode abstraite**

Les méthodes abstraites sont des méthodes qui n'ont pas de corps et DOIVENT être implémentées dans la classe dérivée.

Elles ne peuvent exister que dans des classes abstraites. D'une certaine manière, une méthode abstraite est comme un contrat. Si vous voulez vous assurer que toute classe qui hérite de votre classe implémente une certaine méthode, vous pouvez déclarer la classe comme une classe abstraite et la méthode comme une méthode abstraite.

Pour déclarer une classe abstraite, il suffit d'ajouter le mot-clé abstract avant la classe de mots-clés comme ceci :

**abstract class MyClass**

**{**

**}**

Pour déclarer une méthode abstraite à l'intérieur d'une classe abstraite, ajoutez le mot-clé abstract avant le type de retour, comme ceci :

**public abstract void MyAbstractMethod() ;**

Comme les méthodes abstraites n'ont pas de corps, nous terminons la déclaration par un point-virgule.

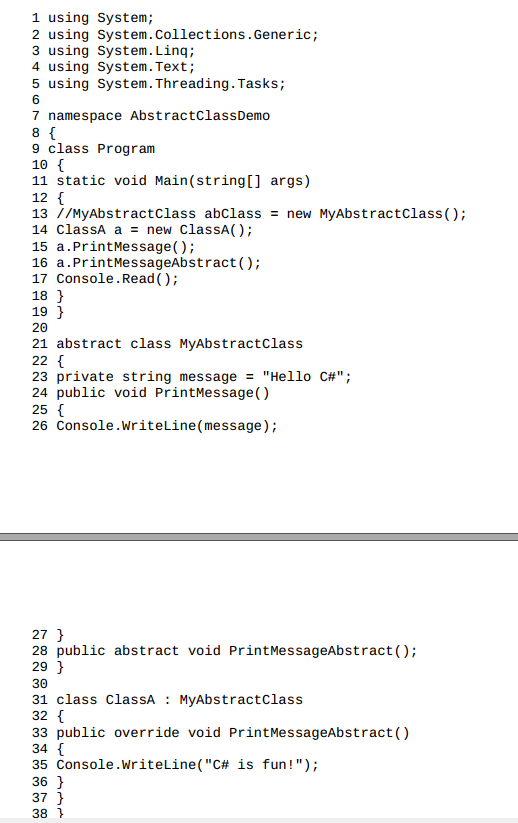
Pour implémenter une méthode abstraite dans la classe dérivée, nous utilisons le mot-clé override, comme ceci.

Public ovveride void MyAbstractMethod ()

{

}

Le code ci-dessous montre un exemple de classe abstraite.



La classe abstraite se situe entre les lignes 21 et 29. Elle contient un message de champ privé et une méthode publique PrintMessage(). Elle contient également une méthode abstraite PrintMessageAbstract() à la ligne 28. Les lignes 31 à 37 montrent la classe dérivée qui implémente la méthode abstraite (lignes 33 à 36).

Si vous exécutez le programme ci-dessus, vous obtiendrez

Bonjour C#

Le C# est amusant !

Remarquez que la ligne 13 est commentée avec le signe // ? Si vous supprimez les deux barres obliques, vous obtiendrez une erreur car une classe abstraite ne peut pas être instanciée.

## **Interfaces**

Ensuite, examinons les interfaces. Les interfaces ressemblent beaucoup à des classes abstraites en ce sens qu'elles ne peuvent pas être instanciées et doivent être héritées. Cependant, les interfaces sont plus conceptuelles que les classes abstraites. Elles ne peuvent contenir que des méthodes sans corps.

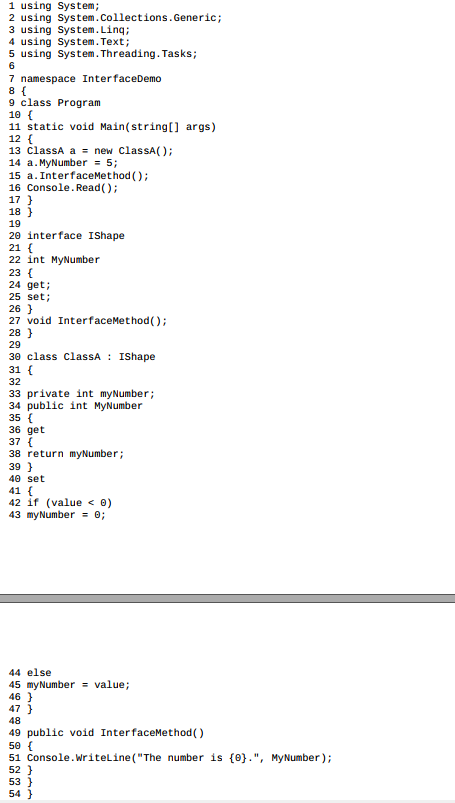
En outre, elles ne peuvent pas contenir de champs mais peuvent contenir des propriétés. Les interfaces ne peuvent pas non plus avoir de membres statiques. Lorsqu'une classe enfant hérite d'une interface, on dit qu'elle implémente l'interface.

L'une des principales différences entre une classe abstraite et une interface est qu'une classe ne peut hériter que d'une seule classe abstraite mais peut implémenter plusieurs interfaces.

Nous ne montrerons pas d'exemple d'implémentation d'interfaces multiples car c'est un sujet avancé qui dépasse la portée de ce livre.

Le code ci-dessous montre un exemple de la façon dont une classe peut implémenter une interface.

Il est courant de commencer le nom d'une interface par la lettre I. Toutes les propriétés et méthodes d'une interface sont publiques, il n'est donc pas nécessaire d'y ajouter des modificateurs d'accès.



L'interface est déclarée sur les lignes 20 à 28. Sur les lignes 22 à 26, on déclare une propriété et sur la ligne 27, on déclare une méthode.

La classe A implémente l'interface IShape. La propriété est implémentée sur les lignes 33 à 47 où nous avons déclaré un champ de soutien privé (myNumber) pour la propriété et mis en œuvre certaines règles de contrôle.

La méthode est mise en œuvre sur les lignes 49 à 52. Nous n'avons pas besoin d'utiliser le mot-clé override lorsque nous implémentons une méthode qui appartient à une interface.

Si vous exécutez ce programme, vous obtiendrez :

The number is 5.

## **Révision des modificateurs d'accès**

Maintenant que nous avons abordé divers sujets liés à l'héritage, examinons à nouveau le concept de modificateurs d'accès dans la programmation orientée objet.

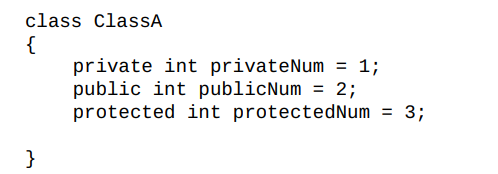
Nous avons appris plus tôt qu'un modificateur d'accès est comme un gardien. Il contrôle qui a accès à un certain champ, une propriété ou une méthode. Le C# est livré avec 4 modificateurs d'accès : private, public, protected et internal.

Tout ce qui est déclaré comme internal n'est accessible que dans l'ensemble en cours.

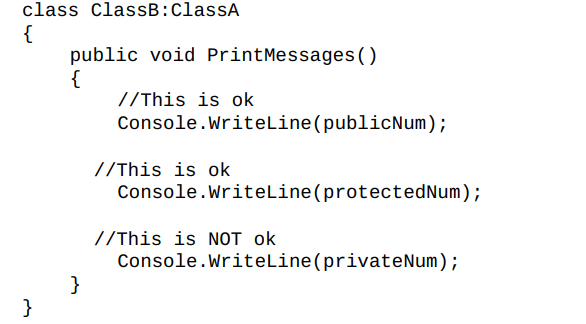
Comme nous ne couvrirons pas les assemblages dans ce livre, nous ne démontrerons pas comment fonctionne internal.

Pour comprendre comment fonctionnent les notions de privé, public et protégé, considérons l'exemple ci-dessous. Nous utiliserons des champs pour démontrer le concept. Il en va de même pour les méthodes et les propriétés.

Supposons que nous ayons une classe avec trois champs :



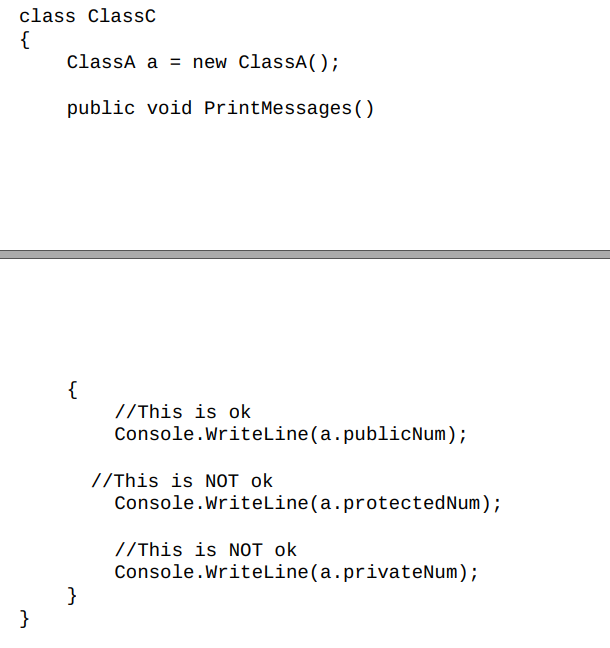
Si la classe B est dérivée de la classe A,



Les deux premières déclarations WriteLine() ne nous donneront aucune erreur car une classe dérivée peut accéder à tous les champs publics et protégés de la classe parent.

Cependant, la troisième instruction nous donne une erreur car privateNum est un champ privé et n'est donc accessible que dans la classe A elle-même.

Si une classe n'est pas dérivée de la ClassA, nous devons instancier un objet ClassA afin d'accéder au champ public de la ClassA. Cependant, même avec un objet ClassA, nous ne pouvons pas accéder aux champs privés et protégés de la ClassA. Dans l'exemple ci-dessous, la ClassC n'est pas dérivée de la ClassA. Par conséquent, la première instruction WriteLine() ne nous donnera aucune erreur, mais les deuxième et troisième instructions le feront.



En bref, tout ce qui est déclaré public est accessible partout ; il n'y a aucune restriction d'accès aux membres publics. En revanche, tout ce qui est déclaré comme privé n'est accessible qu'à l'intérieur de la classe dans laquelle il est déclaré.

Tout ce qui est déclaré comme protégé est accessible au sein de la classe dans laquelle il est déclaré et de toute classe qui en dérive.

# **Chapitre 9 : Enum et Struct**

Dans les chapitres 3 et 4, nous avons examiné certains types de données intégrées fournies par C#. Il s'agit notamment de types de valeurs comme int, float et double et de types de données de référence comme les tableaux, les chaînes de caractères et les listes. De plus, nous avons également examiné comment vous pouvez écrire vos propres classes dans les chapitres 7 et 8. Une classe peut être considérée comme un type de données avancé défini par l'utilisateur qui regroupe un ensemble de champs, de propriétés et de méthodes connexes en une unité logique.

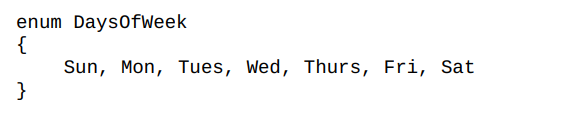
Dans ce chapitre, nous allons examiner deux autres types de données définies par l'utilisateur en C# - enum et struct.

## **Enum**

Un enum (qui signifie type énuméré) est un type de données spécial qui permet aux programmeurs de fournir des noms significatifs pour un ensemble de constantes intégrales.

Pour déclarer un enum, nous utilisons le mot-clé enum suivi du nom de l'enum. Les membres de l'énumération sont entourés d'un ensemble d'accolades et séparés par des virgules.

Un exemple est présenté ci-dessous :



Notez que nous ne mettons pas de point-virgule à la fin du dernier membre.

Après avoir déclaré l'énumération des jours de la semaine, nous pouvons déclarer et initialiser une variable DaysOfWeek comme ceci :

DaysOfWeek myDays = DaysOfWeek.Mon ;

Le nom de la variable est myDays. Si nous écrivons :

Console.WriteLine(myDays) ;

Nous aurons : **Mon**

Par défaut, chaque membre de l'énumération se voit attribuer une valeur entière, en partant de zéro. C'est-à-dire que dans notre exemple, Sun a une valeur de 0, Mon de 1, Tues de 2 et ainsi de suite.

Comme les membres d'une énumération sont essentiellement des entiers, nous pouvons transformer une variable DaysOfWeek en int et vice versa. Par exemple,

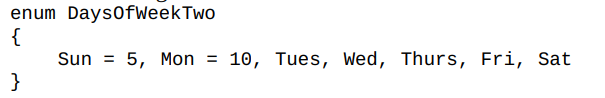
Console.WriteLine((int)myDays) ;

nous donne l'entier 1 alors que

Console.WriteLine((DaysOfWeek)1) ;

nous donne Mon.

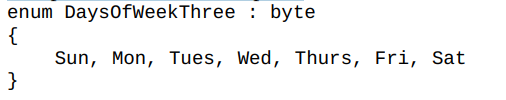
Si vous souhaitez attribuer un ensemble différent d'entiers à vos membres énumérateurs, vous pouvez procéder comme suit



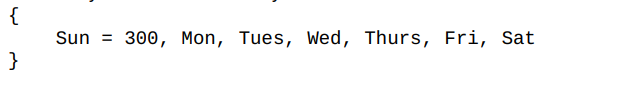
Maintenant, on attribue à Sun une valeur de 5 et à Mon une valeur de 10. Comme nous n'avons pas attribué de valeurs pour les journées du mardi au samedi, les nombres consécutifs après 10 leur seront attribués.

C'est-à-dire Tues = 11, Wed = 12 et ainsi de suite.

Toutes les énumérations sont stockées en interne sous forme de nombres entiers (int). Si vous souhaitez changer le type de données sous-jacent de int à un autre type de données, vous devez ajouter un point après le nom de l'énumération, suivi du type de données souhaité. Tous les types de données entiers sont autorisés à l’exception de char. Un exemple est



Bien entendu, si vous utilisez un type de données en octets, vous ne pouvez pas faire quelque chose comme



car l'intervalle pour l'octet est de 0 à 255.

Il y a deux raisons principales pour utiliser les énumérations. La première est d'améliorer la lisibilité de votre code. L'énoncé

myDays = DaysOfWeek.Mon ;

est plus explicite que la déclaration

myDays = 1 ;

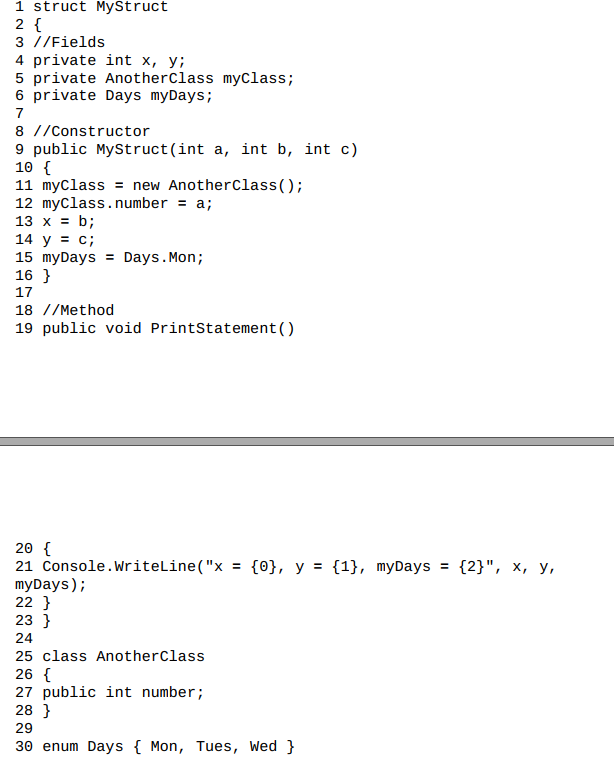
La deuxième raison est de limiter les valeurs que peut prendre une variable. Si nous avons une variable qui stocke les jours d'une semaine, nous pouvons accidentellement lui attribuer la valeur 10. Cela peut être évité lorsque nous utilisons une énumération, car nous ne pouvons affecter à la variable que les membres prédéfinis de l'énumération.

## **Struct**

Examinons maintenant le type de données de struct.

Une structure est similaire à une classe à bien des égards. Comme les classes, elles contiennent des éléments tels que des propriétés, des constructeurs, des méthodes et des champs et vous permettent de regrouper des membres apparentés dans un seul et même paquet afin de les manipuler en tant que groupe.

Pour déclarer une structure, vous utilisez le mot-clé struct. En voici un exemple :

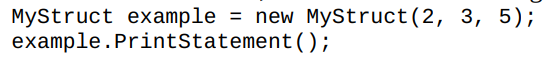


La structure est déclarée des lignes 1 à 23. Sur la ligne 4, nous avons déclaré deux champs d'entrée privés pour la structure. Sur la ligne 5, nous avons déclaré un autre champ privé appelé myClass. Ce champ est une instance de la classe AnotherClass. Sur la ligne 6, nous avons déclaré une variable d'énumération myDays. Les deux champs (myClass et myDays) sont spécialement inclus dans cet exemple pour démontrer comment nous pouvons inclure une instance de classe et une variable d'énumération comme champs d'une structure. Les structures (et les classes) peuvent contenir des variables d'énumération et des instances d'autres structures et classes en tant que champs.

Après avoir déclaré les champs, nous avons déclaré le constructeur de la structure (lignes 9 à 16), suivi d'une méthode pour afficher les valeurs de x, y et myDays. (lignes 19 à 22).

Après avoir déclaré la structure, nous avons déclaré la classe AnotherClass aux lignes 25 à 28 et l'énumération Days à la ligne 30. Dans cet exemple, nous avons déclaré la classe et l'énumération en dehors de la structure myStruct. Cependant, il nous est possible de déclarer l'enum ou la classe à l'intérieur de la structure elle-même. Une énumération, une structure ou une classe peut être imbriquée dans une autre structure ou classe. Nous verrons un exemple d'une énumération déclarée à l'intérieur d'une classe lorsque nous travaillerons sur le projet à la fin du livre.

Pour utiliser la structure ci-dessus, nous pouvons ajouter le code suivant à notre méthode Main() :



Si nous exécutons le code, nous obtiendrons



Il y a deux différences principales entre une structure et une classe. Premièrement, le type de données struct ne prend pas en charge l'héritage. Par conséquent, vous ne pouvez pas dériver une structure d'une autre. Cependant, une structure peut implémenter une interface. La façon de le faire est identique à celle des classes. Reportez-vous au chapitre 8 pour plus d'informations.

La deuxième différence entre les structures et les classes est que les structures sont des types de valeur alors que les classes sont des types de référence.

Pour une liste complète des différences entre une structure et une classe, consultez la page suivante : <https://msdn.microsoft.com/en-us/library/saxz13w4.aspx>

# **Chapitre 10 : LINQ**

LINQ "Language-Integrated Query" signifie **Requête Intégrée au langage** et est une fonctionnalité intéressante du C# qui vous permet d'interroger les données de votre programme. Dans ce chapitre, nous couvrirons une brève introduction à LINQ suivie de deux exemples d'utilisation de LINQ.

Commençons par apprendre comment écrire une requête LINQ. La syntaxe typique d'une requête LINQ est

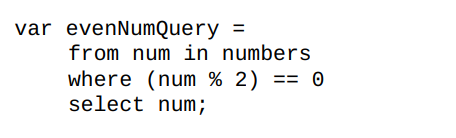


Supposons que nous ayons un tableau de nombres et que nous voulions sélectionner tous les nombres pairs du tableau. Nous pouvons le faire facilement avec LINQ.

Tout d'abord, déclarons le tableau.



Ensuite, nous écrivons une requête LINQ comme suit :



La requête se fait de la deuxième à la quatrième ligne. Les lecteurs qui ont une expérience de SQL trouveront probablement cette requête assez familière. La requête se compose de trois parties. La première partie



indique que nous effectuons la requête sur le tableau des nombres. num est le nom que nous utilisons pour représenter les différents éléments du tableau. La ligne suivante



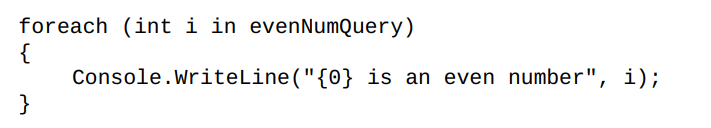
teste les différents éléments pour déterminer si le reste de num divisé par 2 est égal à zéro. Si c'est le cas, num est un nombre pair. La troisième ligne



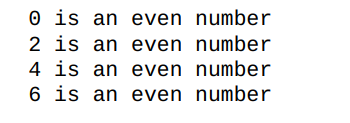
sélectionne tous les éléments qui répondent à ce critère.

Ce résultat est ensuite attribué à la variable evenNumQuery, qui est déclarée de type var. var est un type de données spécial que nous utilisons chaque fois que nous voulons que le compilateur détermine le type de données lui-même. Ceci est nécessaire car dans notre exemple, le type de données de evenNumQuery est assez complexe ; nous ferions mieux de laisser C# déterminer le type de données pour nous.

Après avoir créé l'instruction de requête, nous pouvons exécuter la requête en écrivant



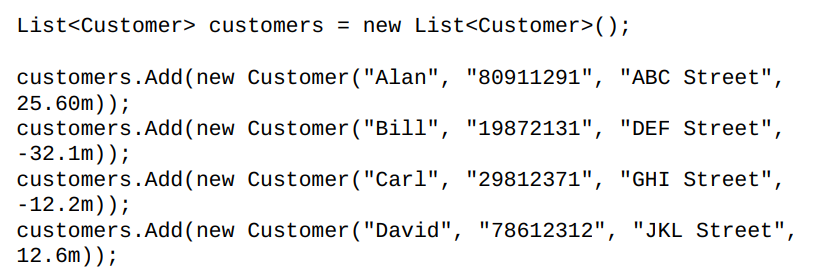
Si vous exécutez ce code, vous obtiendrez



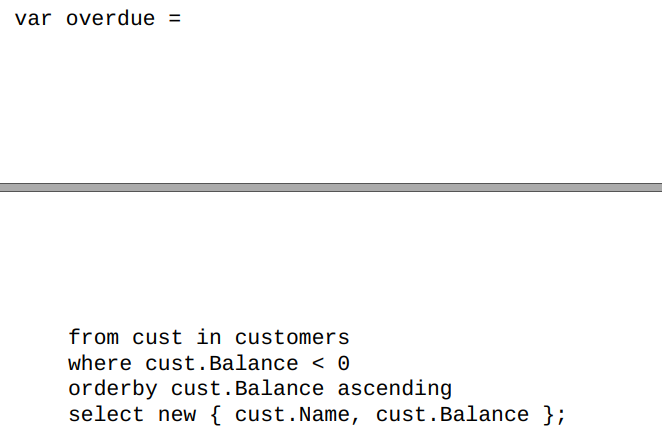
C'est tout. C'est dire à quel point il est facile d'utiliser LINQ. Passons maintenant à un exemple plus complexe de LINQ.

Supposons que vous ayez une classe de client avec comme propriétés Nom, Téléphone, Adresse et Solde et un constructeur pour initialiser chacune de ces propriétés.

Nous pouvons créer une liste d'objets Customer dans notre méthode Main() en utilisant le code ci-dessous.



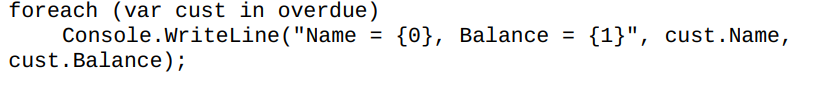
Supposons maintenant que nous voulions rechercher tous les clients dont le solde de compte est négatif, nous pouvons utiliser la requête LINQ suivante.



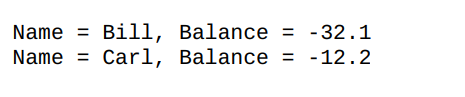
Cette requête est similaire à la première, avec deux différences principales. Ici, nous avons utilisé deux mots clés supplémentaires, **orderby** et **ascending**, pour classer les résultats par ordre croissant.

En outre, nous avons utilisé le nouveau mot-clé dans l'énoncé de sélection. Le nouveau mot-clé est nécessaire chaque fois que nous voulons sélectionner plus d'un champ dans les objets.

Pour exécuter et afficher les résultats, nous pouvons utiliser la boucle foreach ci-dessous :



Nous obtiendrons :



# **Chapitre 11 : Traitement des dossiers**

Cool ! Nous sommes arrivés au dernier chapitre du livre avant le projet. Dans ce chapitre, nous allons apprendre à lire et à écrire dans un fichier externe.

Dans le chapitre 5, nous avons appris comment obtenir des informations des utilisateurs en utilisant la méthode ReadLine(). Cependant, dans certains cas, il n'est pas pratique d'amener les utilisateurs à entrer des données dans notre programme, surtout si notre programme doit travailler avec de grandes quantités de données. Dans de tels cas, il est plus pratique de préparer les informations nécessaires sous la forme d'un fichier externe et de faire en sorte que nos programmes lisent les informations contenues dans ce fichier.

Le C# nous fournit un certain nombre de classes pour travailler avec les fichiers. Les classes que nous allons examiner dans ce chapitre sont les classes File, StreamWriter et StreamReader. Ces trois classes sont disponibles dans l'espace de noms System.IO. Pour utiliser les méthodes de ce chapitre, vous devez ajouter la directive

**using System.IO ;**

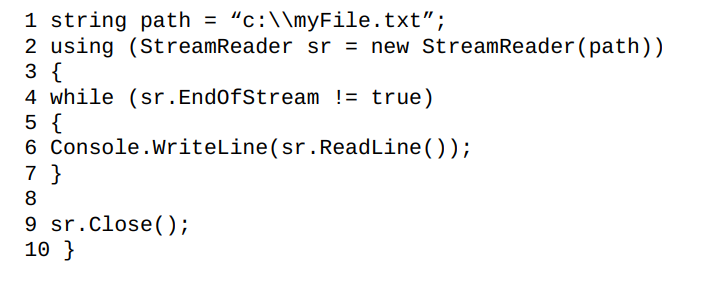
au début de votre code.

## **Lecture d'un fichier texte**

Pour lire les données d'un fichier texte, nous utilisons la classe StreamReader.

Supposons que nous voulions lire les données du fichier "monFichier.txt" situé sur le lecteur C.

L'exemple ci-dessous montre comment le faire.



Sur la ligne 1, nous déclarons d'abord un chemin de variable de chaîne de caractères et nous attribuons le chemin du fichier à la variable.



Notez que nous devons utiliser des doubles barres obliques \\ lors de l'écriture du chemin. En effet, si nous n'utilisons qu'une seule barre oblique, le compilateur pensera que la simple barre oblique est le début d'une séquence d'échappement et interprétera \m comme une séquence d'échappement.

Cela entraînera une erreur.

Sur la ligne 2, nous créons une instance de StreamReader. Le constructeur StreamReader prend en argument le chemin du fichier à lire.

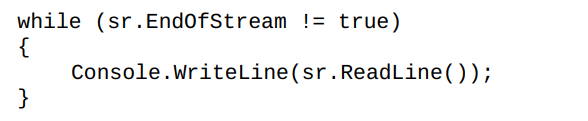


Remarquez que nous créons cette instance de StreamReader à l'intérieur d'une paire de parenthèses qui suit le mot "using" sur la ligne 2 ?

Le mot-clé using ici est différent de celui que nous utilisons lors de la rédaction d'une directive.

Le mot-clé using ici assure que la méthode Dispose() est toujours appelée. La méthode Dispose() est une méthode pré-écrite dans l'espace de noms du système qui ferme ou libère toutes les ressources non gérées telles que les fichiers et les flux lorsqu'elles ne sont plus nécessaires. Lorsque nous utilisons le mot-clé using, nous nous assurons que la méthode Dispose() est appelée même si une exception se produit et empêche notre code d'atteindre la ligne 9 où nous fermons manuellement le fichier. Il est recommandé de toujours utiliser le mot-clé using lorsque vous traitez des fichiers. Le code pour lire et fermer le fichier est entouré d'accolades { } après la déclaration d'utilisation.

Des lignes 4 à 7, nous utilisons une boucle de temps pour lire le fichier texte ligne par ligne.



EndOfStream est une propriété de la classe StreamReader qui retourne vrai lorsque la fin du fichier est atteinte. Tant que la fin du fichier n'est pas atteinte, la boucle while continuera à fonctionner.

Dans la boucle while, nous avons la déclaration suivante



sr.ReadLine() lit une ligne du fichier texte et la renvoie sous forme de chaîne de caractères. Cette chaîne est ensuite affichée à l'écran à l'aide de la méthode Console.WriteLine().

Enfin, une fois la lecture du fichier terminée, nous fermons le fichier afin que d'autres programmes puissent l'utiliser. Vous devez toujours fermer votre fichier dès que vous n'en avez plus besoin.

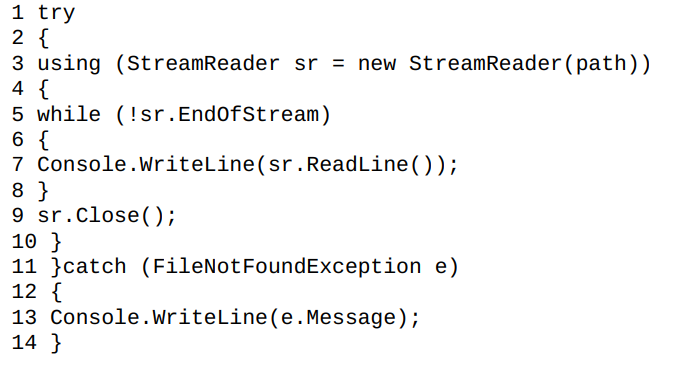


C'est tout. C'est comme ça qu'on lit un fichier texte en C#. C'est assez simple, non ?

Cependant, il y a un problème avec le code ci-dessus. Ce code génère une erreur si le fichier "monFichier.txt" est introuvable. Nous avons deux options ici.

### **Option 1 : try...catch**

La première option consiste à utiliser une déclaration "Essayez...Détectez" comme indiqué ci-dessous :



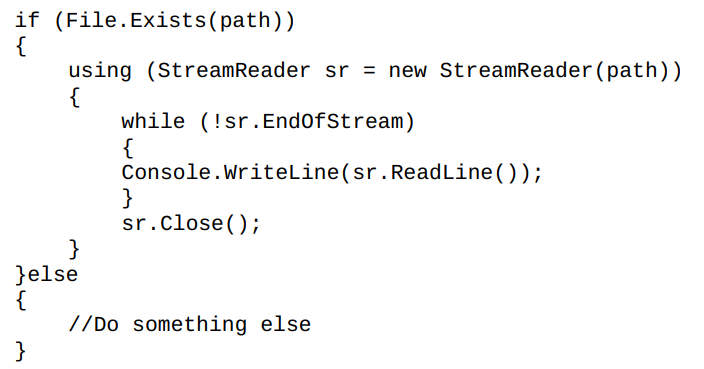
Des lignes 1 à 11, nous essayons d'ouvrir, de lire et de fermer le fichier dans le bloc d'essai.

De la ligne 11 à la ligne 14, nous utilisons un bloc d'essai pour détecter l'exception FileNotFoundException si le fichier n'est pas trouvé. Dans le bloc de capture, nous affichons une déclaration d'erreur pour informer les utilisateurs que le fichier n'est pas trouvé.

### **Option 2 : File.Exists()**

La deuxième méthode pour traiter un scénario de "fichier non trouvé (file not found)" consiste à utiliser la méthode Exists() dans la classe File. Comme son nom l'indique, la méthode Exists() vérifie si un fichier existe. La classe File est une classe pré-écrite dans l'espace de noms System.IO qui fournit des méthodes statiques pour la création, la copie, la suppression, le déplacement et l'ouverture d'un seul fichier.

Pour utiliser la méthode Exists(), nous utilisons une instruction if pour vérifier si le fichier existe avant d'utiliser un StreamReader pour ouvrir et lire le fichier.



Dans le bloc "sinon", nous pouvons écrire un code pour créer le fichier s'il n'est pas trouvé.

Comme vous pouvez le voir, les deux méthodes pour traiter les cas où le fichier est manquant sont assez similaires. Cependant, la méthode File.Exists() est la méthode préférée car elle est plus rapide que la déclaration try...catch.

### **Ecrire dans un fichier texte**

Ensuite, voyons comment écrire dans un fichier texte.

Pour écrire dans un fichier texte, nous utilisons la classe StreamWriter.

Si vous voulez ajouter des données à un fichier existant, vous créez une instance de StreamWriter comme ceci



où path est le chemin du fichier et true indique que nous voulons ajouter les données.

Si vous souhaitez écraser des données existantes dans le fichier, vous devez créer une instance de StreamWriter comme ceci



Lorsque nous créons l'instance de StreamWriter, le constructeur recherche le fichier au niveau du chemin donné. Si le fichier n'est pas trouvé, il crée le fichier.

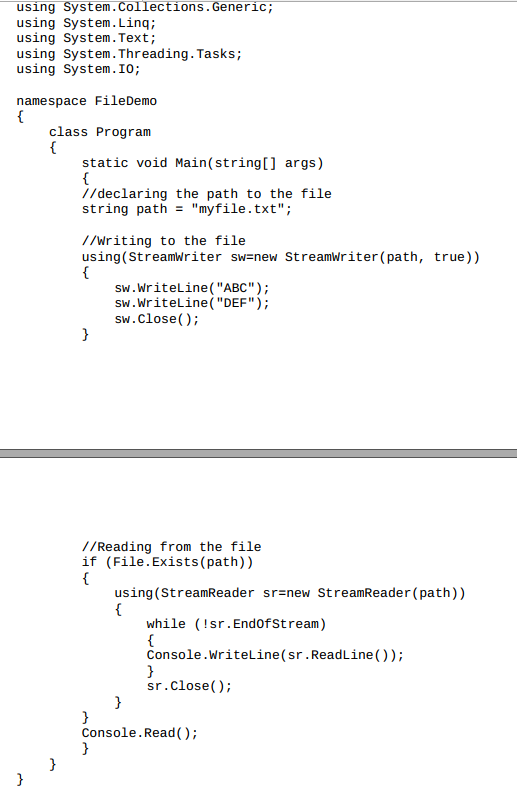
Après avoir instancié notre objet StreamWriter, nous pouvons commencer à écrire dans notre fichier en utilisant la méthode WriteLine() comme indiqué ci-dessous :



Une fois que nous avons fini d'écrire dans le fichier, nous devons fermer le fichier en écrivant



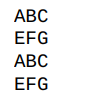
Notez que lorsque vous écrivez dans un fichier texte, il est également bon de joindre votre code dans une déclaration d'utilisation. Le code ci-dessous montre un exemple complet de la manière dont tous ces éléments se combinent.



Dans cet exemple, nous choisissons d'ajouter des données à notre fichier lorsque nous y écrivons. Lorsque vous exécutez ce programme pour la première fois, vous obtenez



comme la sortie du fichier et l'affichage à l'écran. Si vous l'exécutez pour la deuxième fois, vous obtiendrez



Comme le chemin complet de "myfile.txt" n'est pas donné dans cet exemple, le fichier texte sera créé dans le même dossier que le fichier .exe, qui se trouve dans le dossier FileDemo > FileDemo > Bin > Debug.

# **Projet - Un logiciel de paie simple**

Félicitations !

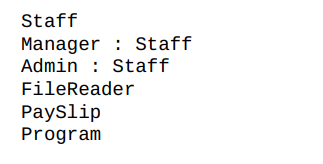
Nous avons maintenant terminé les concepts de base en C#. Dans ce dernier chapitre, nous allons nous mouiller les pieds en codant une application console complète qui génère les fiches de salaire d'une petite entreprise.

Vous êtes prêts ?

## **Vue d'ensemble**

Tout d'abord, créons une nouvelle application de console et appelons - la CSProject.

Cette application se compose de six classes, comme indiqué ci-dessous.



La classe Staff contient des informations sur chaque membre du personnel de l'entreprise. Elle contient également une méthode virtuelle appelée CalculatePay() qui calcule la rémunération de chaque membre du personnel.

Les classes Manager et Admin héritent de la classe Personnel et remplacent la méthode CalculatePay().

La classe FileReader contient une méthode simple qui lit à partir d'un fichier .txt et crée une liste d'objets Staff basée sur le contenu du fichier .txt.

La classe PaySlip génère la fiche de paie de chaque employé de l'entreprise. En outre, elle génère également un résumé des détails du personnel qui a travaillé moins de 10 heures dans un mois.

Enfin, la classe Program contient la méthode Main() qui agit comme point d'entrée principal de notre application.

### **La class Staff**

Tout d'abord, commençons par la classe Staff. La classe Staff contient des informations de base sur un employé et fournit une méthode de calcul de la rémunération de base. Elle sert de classe mère dont seront dérivées deux autres classes.

#### **Champs**

Cette classe dispose d'un champ float (flottant) privé appelé hourlyRate (taux horaire) et d'un champ int privé appelé hWorked (heure travaillée). Essayez de déclarer ces champs vous-même.

#### **Propriétés**

Ensuite, déclarez trois propriétés publiques auto-implémentées pour la classe. Ces propriétés sont TotalPay, BasicPay et NameOfStaff.

TotalPay (salaire total) est une propriété flottante et dispose d'un setter protégé. BasicPay (Salaire de base) est une propriété flottante et possède un paramètre privé. NameOfStaff (nom du personnel) est une propriété de type chaîne de caractères et possède un paramètre privé. Les getters des trois propriétés sont publics. Il n'est donc pas nécessaire de déclarer les modificateurs d'accès de ces getters car ils ont le même niveau d'accès que les propriétés.

En plus de ces trois méthodes automatiques, la classe Staff dispose également d'une propriété public appelé HoursWorked. Le champ de soutien de cette propriété est le champ hWorked.

Cette propriété a un getter qui renvoie simplement la valeur de hWorked. Le setter vérifie si la valeur définie pour HoursWorked est supérieure à 0, et si c'est le cas, il attribue une valeur à hWorked. Si ce n'est pas le cas, il attribue la valeur 0 à hWorked. Essayez de déclarer cette propriété vous-même. Vous pouvez vous référer au chapitre 7 pour obtenir de l'aide.

#### **Constructeur**

La classe Staff dispose d'un constructeur public avec deux paramètres, le nom->name (chaîne->string) et le taux->rate (flottant->float). À l'intérieur du constructeur, nous attribuons les deux paramètres à la propriété NameOfStaff et au champ hourlyRate respectivement. Essayez de coder ce constructeur vous-même.

#### **Méthode**

Maintenant, écrivons les méthodes pour la classe.

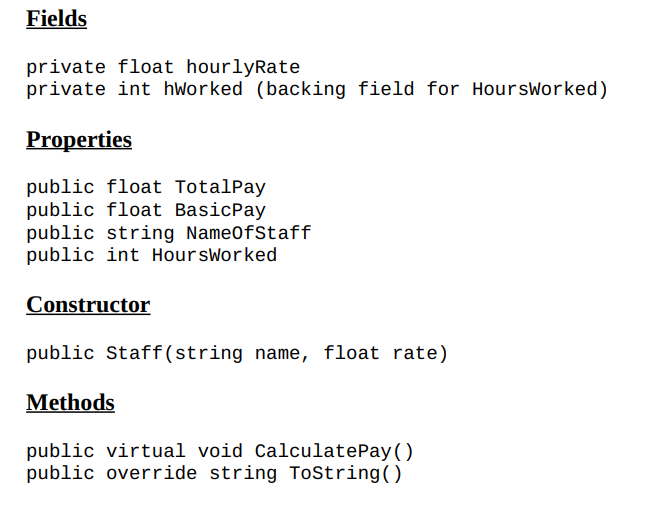
Tout d'abord, nous allons coder une méthode virtuelle appelée CalculatePay().

CalculatePay() est publique, n'a pas de paramètres et ne retourne pas de valeur. La méthode fait trois choses :

Premièrement, elle affiche la ligne "Calculer la rémunération..." sur l'écran. Ensuite, elle attribue la valeur de hWorked\*hourlyRate à la propriété BasicPay. Enfin, elle attribue la valeur de BasicPay à la propriété TotalPay. En d'autres termes, BasicPay et TotalPay auront la même valeur. Essayez de coder cette méthode vous-même.

Enfin, écrivez une méthode ToString() pour afficher les valeurs des champs et des propriétés de la classe Staff. C'est tout ce qu'il y a dans la classe Staff.

Le tableau ci-dessous présente un résumé de la classe Staff.



### **La classe Manager (Manager : Staff)**

Ensuite, passons au codage de la classe Manager.

#### **Champs**

La classe Manager est une classe enfant de la classe Staff. Elle possède un champ de type const privé appelé managerHourlyRate qui est de type flottant. Essayez de déclarer ce champ et de l'initialiser avec une valeur de 50.

#### **Propriétés**

Manager a également une propriété publique auto-implémentée appelée Allowance(Indemnité).

Allowance est de type int et possède un setter privé. Essayez de coder cette propriété.

#### **Constructeur**

Maintenant, déclarons le constructeur pour Manager. La classe Manager a un constructeur public avec un paramètre de type chaîne, le nom.

La tâche du constructeur consiste à appeler le constructeur de base et à lui transmettre le nom du paramètre et le champ managerHourlyRate au constructeur de base. À part cela, le constructeur enfant ne fait rien. Par conséquent, il n'y a rien dans les accolades du constructeur enfant. Essayez de coder ce constructeur vous-même. Vous pouvez vous référer au résumé de la classe Manager ci-dessous pour vous aider si vous avez des problèmes pour coder le constructeur.

#### **Méthode**

Ensuite, codons une méthode pour remplacer la méthode CalculatePay() dans la classe Staff. Comme Manager est dérivé de Staff, il a accès aux propriétés BasicPay, TotalPay et HoursWorked déclarées dans la classe Staff.

En outre, Manager a également sa propre propriété – Allowance(indemnité). Nous utiliserons ces quatre propriétés dans le cadre de cette méthode.

Tout d'abord, déclarons la méthode. CalculatePay() est publique et ne renvoie aucune valeur. Nous devons utiliser le mot-clé override lors de la déclaration de cette méthode car il remplace la méthode CalculatePay() dans la classe Staff.

Le mot-clé **virtuel** est utilisé pour modifier une méthode, une propriété, un indexeur ou un événement déclaré dans la classe de base et permettre de le remplacer dans la classe dérivée. Le mot-clé **override** est utilisé pour étendre ou modifier une méthode, une propriété, un indexeur ou un événement virtuel/abstrait de la classe de base dans la classe dérivée.

Dans la méthode CalculatePay() de la classe Manager, nous allons d'abord appeler la méthode CalculatePay() de la classe parent et l'utiliser pour définir les valeurs de BasicPay et TotalPay. Pour appeler une méthode virtuelle dans la classe parent, il faut utiliser le mot-clé base. Ajoutez la ligne suivante à votre méthode CalculatePay().

base.CalculatePay() ;

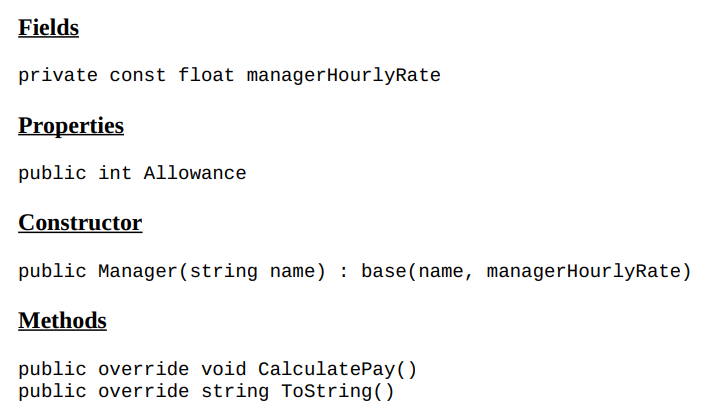
Cette méthode appelle la méthode CalculatePay() dans la classe (parent) de base, qui définit les valeurs de BasicPay et TotalPay. Après avoir défini les valeurs de ces deux propriétés, passons à la définition de la valeur de l’indemnité (Allowance). Nous allons fixer la valeur à 1000.

Ensuite, nous voulons modifier la valeur de TotalPay. D'après la méthode CalculatePay() de la classe de base, TotalPay est égal à BasicPay, qui sont tous deux égaux au produit de hWorked et de hourlyRate.

Cependant, dans la classe enfant Manager, nous voulons mettre à jour la valeur de TotalPay en y ajoutant une indemnité. Supposons qu'un manager reçoive une indemnité de 1 000 $ s'il a travaillé plus de 160 heures au cours de ce mois. Essayez d'utiliser une instruction if pour mettre à jour la valeur de TotalPay sur la base de la valeur des heures travaillées.

Après avoir mis à jour la valeur de TotalPay, la méthode CalculatePay() est terminée.

Enfin, nous devons coder la méthode ToString() pour la classe Manager. Essayez de coder cette méthode. Une fois que vous avez terminé, la classe Manager est complète. Le tableau ci-dessous présente un résumé de la classe Manager.



### **La classe Admin (Admin : Staff)**

La classe suivante est la classe Admin qui est également dérivée de la classe Staff.

#### Champs

La classe Admin dispose de deux champs de constantes privés : overtimeRate (tauxHeureSupplementaire) et adminHourlyRate (tauxHoraireAdministrateur). Les deux champs sont de type float. Essayez de déclarer ces deux champs et de les initialiser avec les valeurs 15,5 et 30 respectivement.

#### **Propriété**

Ensuite, essayez de déclarer une propriété publique qui implémente automatiquement, les heures supplémentaires.

Les heures supplémentaires sont de type flottant et ont un setter privé.

#### **Constructeur**

Maintenant, déclarons le constructeur. Tout comme le constructeur de la classe Manager, le constructeur de la classe Admin est public et possède un paramètre de chaîne de caractères, name. Son travail consiste à appeler simplement le constructeur de base et à lui transmettre le nom du paramètre et le champ adminHourlyRate.

#### Méthode

Enfin, nous sommes prêts à coder la méthode CalculatePay() pour la classe Admin. La méthode CalculatePay() de la classe Admin est très similaire à la méthode de la classe Manager. Commençons par déclarer la méthode.

Ensuite, à l'intérieur des accolades, nous utilisons la méthode CalculatePay() de la classe de base pour définir les propriétés BasicPay et TotalPay d'un personnel administratif.

Après avoir défini les valeurs de ces deux propriétés, nous vérifions si le nombre d'heures travaillées est supérieur à 160. Si c'est le cas, nous mettrons à jour la valeur de la propriété TotalPay.

Supposons qu'un membre du personnel administratif perçoive une rémunération pour les heures supplémentaires en plus de sa rémunération de base s'il/elle a travaillé plus de 160 heures. Essayez d'utiliser une déclaration if pour mettre à jour la propriété TotalPay d'un membre du personnel administratif.

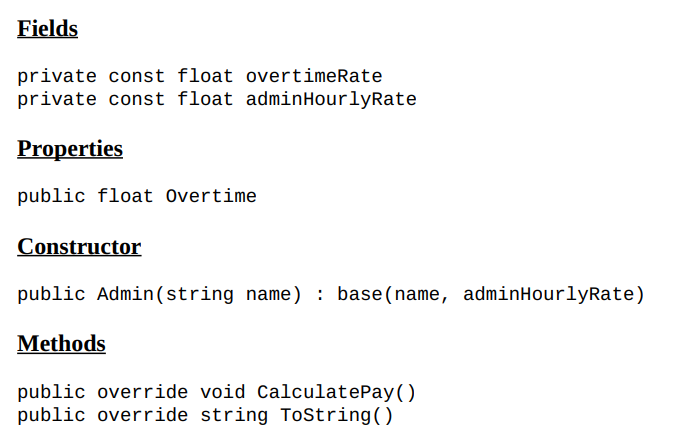
La rémunération des heures supplémentaires est calculée selon la formule suivante



où overtimeRate est un champ privé dans la classe Admin et Overtime (heures supplémentaires) est une propriété dans la même classe. HoursWorked (heures travaillées) est une propriété héritée de la classe Staff.

Fait ?

Super ! Maintenant, passez au code de la méthode ToString(). Avec cela, la classe Admin est complète. Le tableau ci-dessous présente un résumé de la classe.



### **La classe FileReader**

Maintenant, nous sommes prêts à coder la classe FileReader. La classe FileReader est relativement simple.

Elle consiste en une méthode publique appelée ReadFile() qui n'a pas de paramètre. La méthode renvoie une liste d'objets Staff. La déclaration de la méthode est la suivante :

public List<Staff> ReadFile()

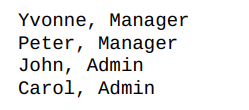
{

}

La méthode ReadFile() lit à partir d'un fichier .txt qui contient les noms et les postes du personnel. Le format est :

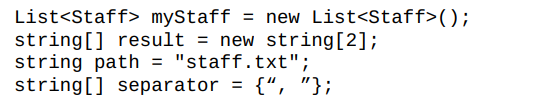
Nom du personnel, poste du personnel

En voici un exemple :



Le nom du fichier texte est "staff.txt" et est stocké dans le même dossier que le fichier .exe. Créez ce fichier vous-même en utilisant Notepad et stockez-le dans le dossier CSProject > CSProject > Bin > Debug où se trouve le fichier .exe.

Maintenant, nous pouvons commencer à coder la méthode ReadFile(). Nous commençons par déclarer quatre variables locales appelées myStaff, result, path et separator comme indiqué ci-dessous.



Ensuite, nous vérifions si le fichier "staff.txt" existe en utilisant une déclaration if et la méthode File.Exists(). Vous devez ajouter la directive en utilisant System.IO ;

afin d'utiliser la méthode File.Exists().

Si le fichier existe, nous utilisons un objet StreamReader pour lire le fichier texte ligne par ligne. Chaque fois que nous lisons une ligne, nous utilisons la méthode Split() (voir le chapitre 4) pour diviser la ligne en deux parties et stocker le résultat dans le tableau des résultats. Par exemple, lorsque nous lisons la première ligne, la méthode Split() la divise en deux chaînes de caractères "Yvonne" et "Manager". Ainsi, result[0] = "Yvonne" et le result[1] = "Manager".

Sur la base de la valeur de result[1], nous utilisons une instruction if pour créer un objet Manager si la valeur de result[1] est "Manager" ou un objet Admin si la valeur est "Admin". Nous ajoutons ces objets à la liste myStaff.

Une fois la lecture du fichier terminée, nous fermons le fichier en utilisant la méthode Close().

Si le fichier n'existe pas, nous affichons un message pour informer les utilisateurs de l'erreur.

Enfin, nous renvoyons la liste myStaff en l'appelant après la déclaration if-else.

C'est tout ce qu'il y a dans la classe FileReader. Nous n'avons pas besoin de déclarer un constructeur pour cette classe. Nous utiliserons simplement le constructeur par défaut que C# crée automatiquement pour nous. Le résumé de la classe FileReader est présenté ci-dessous :

##### **Méthodes**

public List<Staff> ReadFile() ;

### **La classe PaySlip (bulletins de paie)**

Maintenant, codons la classe des bulletins de paie. Cette classe est légèrement différente des autres classes que nous avons vues jusqu'à présent. Outre les champs, les propriétés, les méthodes et les constructeurs, la classe des bulletins de paie comporte également une énumération appelée MonthsOfYear.

#### **Champs**

Tout d'abord, déclarons les champs. La classe a deux champs d'entiers privés nommés mois(month) et année(year). Essayez de les déclarer.

#### **Enum**

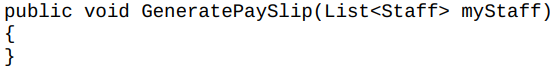
Ensuite, nous déclarons une énumération appelée " MonthsOfYear (Mois de l'année)" dans la classe PaySlip. Mois de l'année représente les douze mois de l'année, où JAN = 1, FEB = 2 etc. Essayez de déclarer cette énumération vous-même. Vous n'avez pas besoin de spécifier de modificateur d'accès pour cette énumération. Une énumération déclarée à l'intérieur d'une classe est privée par défaut.

#### Constructeur

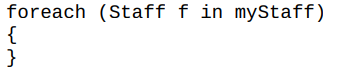
Maintenant, essayez d'ajouter un constructeur à la classe PaySlip. Le constructeur est public et possède deux paramètres int : payMonth et payYear. Dans le constructeur, nous attribuons les deux paramètres aux champs privés month(mois) et year(année) respectivement.

#### **Méthodes**

Ensuite, codons la méthode GeneratePaySlip(). Cette méthode prend en compte une liste d'objets Staff et ne retourne rien. La déclaration de la méthode est



A l'intérieur de la méthode, nous déclarons une variable de chaîne de caractères appelée path. Ensuite, toujours dans la méthode GeneratePaySlip(), nous utilisons une boucle foreach pour parcourir les éléments de myStaff. Cela peut être fait de la manière suivante :



Tout ce qui en découle pour la méthode GeneratePaySlip() doit être codé à l'intérieur des {} de la boucle foreach.

Tout d'abord, nous attribuons une valeur à la variable path en fonction du nom du personnel.

Rappelez-vous que la classe Staff a une propriété appelée NameOfStaff ?

Supposons que NameOfStaff = "Yvonne", nous voulons attribuer la chaîne "Yvonne.txt" à la variable path.

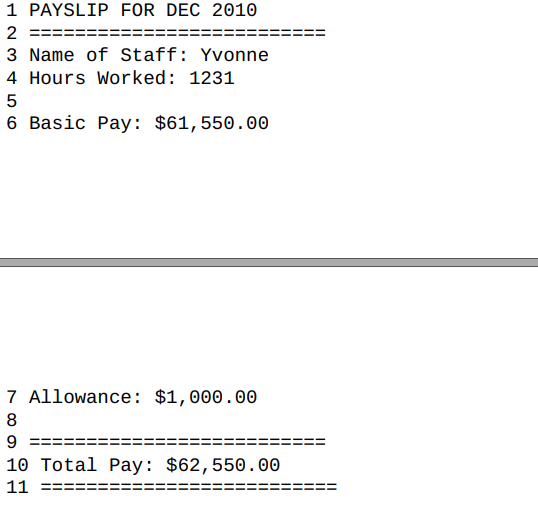
Comment feriez-vous cela ? Essayez de la coder vous-même. (Conseil : vous pouvez utiliser f.NameOfStaff pour accéder au nom du personnel et utiliser l'opérateur + pour concaténer l'extension ".txt")

Après avoir attribué une valeur à path, nous voulons instancier un objet StreamWriter pour écrire dans le fichier au chemin spécifié par la variable path, en écrasant tout contenu existant sur le fichier afin que chaque bulletin de paie généré ne contienne pas de contenu du mois précédent. Reportez-vous au chapitre 11 si vous avez oublié comment

pour utiliser la classe StreamWriter. Appelons l'objet StreamWriter sw.

Nous pouvons ensuite utiliser une série d'états sw.WriteLine() pour générer la fiche de paie de chaque employé.

Une fiche de paie typique pour un manager ressemble à ceci :



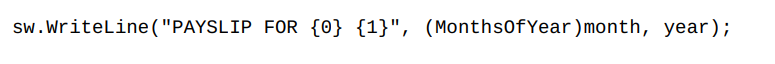
Les numéros à gauche sont ajoutés à titre de référence et ne font pas partie du bulletin de paie proprement dit.

Une fiche de paie typique pour un membre du personnel administratif se présente de la même manière, à l'exception de la ligne 7. Pour un membre du personnel administratif, la ligne 7 se lira comme suit



Voyons maintenant comment générer cette fiche de salaire.

Pour écrire la ligne 1, nous devons accéder aux champs du mois et de l'année dans la classe. Comme le mois est un entier, nous devons le transformer en une valeur de l'énumération des mois de l'année afin qu'il s'écrive DEC au lieu de 12. La déclaration ci-dessous montre comment la ligne 1 peut être écrite.



La ligne 2 est facile à écrire. Elle est simplement constituée d'une série de signes égaux (=). Essayez de la coder vous-même.

Pour écrire les lignes 3 et 4, nous devons accéder aux propriétés NameOfStaff et HoursWorked dans la classe Staff. La déclaration ci-dessous montre comment faire pour la ligne 3.



Essayez de coder vous-même la ligne 4.

Ensuite, nous utilisons une instruction sw.WriteLine("") ; pour imprimer une ligne vide.

Pour écrire la ligne 6, nous devons accéder à la propriété BasicPay dans la classe Staff. En outre, nous devons également utiliser le spécificateur C pour afficher la propriété BasicPay en notation monétaire (voir le chapitre 5). Essayez-le-vous-même.

La ligne 7 est plus difficile car nous devons déterminer le type d'exécution de l'objet courant dans la boucle foreach. Nous avons appris à faire cela au chapitre 8. Si l'instance courante est un objet Manager, nous accédons et affichons la propriété Allowance dans la classe Manager. Pour accéder à la propriété Allowance dans la classe Manager, nous devons lancer f dans un objet Manager en écrivant



Si l'instance actuelle est un objet Admin, nous accédons et affichons la propriété Overtime(Heures supplémentaires) dans la classe Admin. Essayez de coder vous-même la ligne 7.

La ligne 8 est une autre ligne vide et la ligne 9 est constituée d'une série de signes égaux. La ligne 10 indique la rémunération totale du personnel actuel, que nous pouvons obtenir à partir de la propriété TotalPay de la classe Staff. Enfin, la ligne 11 est une autre ligne composée de signes égaux. Essayez de coder ces lignes vous-même.

Enfin, après avoir généré la fiche de paie pour chaque membre du personnel, nous devons fermer le fichier en utilisant la méthode sw.Close().

Cela nous amène à la fin de la méthode GeneratePaySlip(). Une fois que vous avez fini de coder cette méthode, nous pouvons passer à la méthode suivante dans la classe PaySlip.

La méthode suivante génère un récapitulatif des salariés ayant travaillé moins de 10 heures au cours de ce mois. Appelons cette méthode GenerateSummary().

Comme la méthode GeneratePaySlip(), la méthode GenerateSummary() est publique, elle prend en compte une liste d'objets du personnel et ne renvoie aucune valeur. Essayez de déclarer cette méthode vous-même.

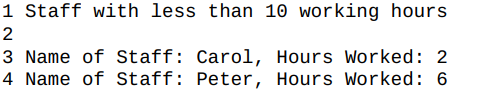
Dans la méthode GenerateSummary(), nous utilisons LINQ pour sélectionner tous les salariés qui ont travaillé moins de 10 heures au cours de ce mois. Nous voulons connaître le nom du personnel et les heures travaillées pour ces employés. En outre, nous voulons classer le résultat par ordre croissant en fonction du nom du personnel. Essayez de coder vous-même cette déclaration LINQ et d'affecter le résultat à une variable var appelée result. Vous pouvez vous référer au chapitre 10 pour obtenir de l'aide.

C'est fait ?

Bien.

Ensuite, déclarons un chemin de variable string et attribuons-lui la chaîne "summary.txt".

Nous sommes maintenant prêts à écrire dans "summary.txt". Déclarons une instance de StreamWriter pour écrire dans ce fichier. Un fichier "summary.txt" typique ressemble à ceci (les numéros à gauche sont ajoutés pour référence) :



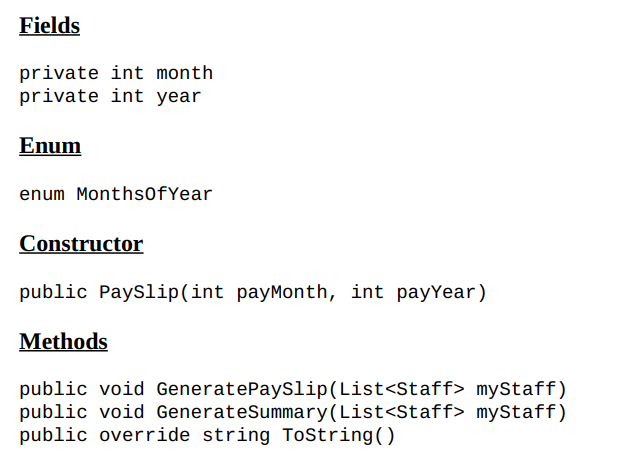
Les lignes 1 et 2 devraient être assez faciles à coder. Essayez de les coder vous-même.

Pour afficher les lignes 3 et 4, nous devons utiliser une boucle foreach pour boucler chaque élément de la variable de résultat obtenue à partir de l'instruction LINQ. Essayez de les coder vous-même.

Après avoir affiché le résultat, vous pouvez fermer le fichier "summary.txt" en utilisant la méthode Close().

C'est tout pour notre méthode GenerateSummary().

Après avoir codé la méthode GenerateSummary(), il nous suffit de coder la méthode ToString() et notre classe PaySlip est terminée. Le tableau ci-dessous présente un résumé de la classe PaySlip.



### **La classe Program**

Nous en arrivons maintenant à la partie la plus importante du projet - la classe Program.

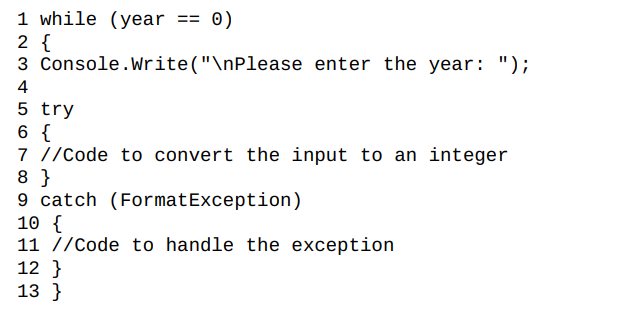
La classe Program n'a qu'une seule méthode - la méthode Main().

#### **La méthode Main()**

Tout d'abord, déclarons quatre variables locales pour la méthode Main(). La première est une liste d'objets Staff. Nous appellerons cette liste myStaff. La suivante est un objet FileReader appelé fr. Les deux autres sont des variables int. Appelons-les month et year et initialisons-les à zéro. Essayez de déclarer vous-même ces variables locales.

Maintenant, nous allons utiliser une boucle while et déclaration d'essai try catch pour inciter les utilisateurs à entrer l'année pour la fiche de paie. La boucle incitera à plusieurs reprises les utilisateurs à entrer l'année jusqu'à ce qu'elle obtienne une valeur valide.

Pour ce faire, nous utilisons la boucle while ci-dessous :



Dans le bloc d'essai (ligne 7), nous lisons la valeur que l'utilisateur a entrée et essayons de la convertir en un nombre entier. Nous l'affectons ensuite à la variable année. Si la conversion est réussie, l'année ne sera plus zéro et la boucle while sortira. Essayez de coder vous-même le bloc d'essai.

Si la conversion ne réussit pas, nous captons l'erreur dans le bloc catch pour éviter que le programme ne plante. Essayez de coder un message d'erreur dans le bloc de saisie (ligne 11). Si la conversion échoue, l'année reste à zéro et la boucle d'attente continue. Les utilisateurs seront invités à plusieurs reprises à entrer l'année jusqu'à ce qu'ils entrent une valeur valide.

Une fois que vous avez terminé avec ce bloc "while", vous pouvez passer au codage du bloc "while" pour inviter les utilisateurs à entrer le mois. Le bloc "while" pour la variable "month" est très similaire à celui de la variable "year". Toutefois, nous souhaitons effectuer davantage de vérifications pour la variable du mois.

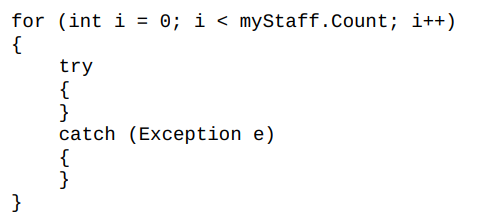
Dans le bloc try, nous essayons d'abord de convertir l'entrée en un nombre entier et de l'affecter à la variable month. Si cela réussit, nous utilisons une déclaration if pour vérifier si le mois est inférieur à 1 ou supérieur à 12. Si c'est le cas, l'entrée n'est pas valide. Nous affichons un message d'erreur pour informer les utilisateurs qu'ils ont saisi une valeur non valable. En outre, nous remettons également le mois à zéro afin que la boucle while se répète. Essayez de coder vous-même ce bloc d'essai.

Après avoir codé le bloc d'essai, vous pouvez procéder au codage du bloc de capture qui informe simplement les utilisateurs de l'erreur.

C'est fait ? Bien.

Ensuite, nous ajouterons des éléments à notre liste myStaff. Nous le faisons en utilisant l'objet fr pour appeler la méthode ReadFile() dans la classe FileReader et en affectant le résultat à myStaff.

Nous pouvons alors commencer à calculer la rémunération de chaque membre du personnel. Pour cela, nous utiliserons la boucle suivante.



Dans la boucle for, nous utilisons une déclaration try catch. Dans le bloc try, nous faisons ce qui suit :

Tout d'abord, nous demandons à l'utilisateur d'entrer le nombre d'heures travaillées pour chaque employé. Voici un exemple d'invite

Entrez les heures travaillées pour Yvonne :

où "Yvonne" est le nom du personnel. Vous devez accéder à la propriété "NameOfStaff" pour chaque personnel en écrivant myStaff[i].NameOfStaff.

Ensuite, on lit la valeur entrée par l’utilisateur, essayez de la convertir en un nombre entier et attribuez-le à la propriété HoursWorked de l'objet Staff.

Ensuite, nous appelons la méthode CalculatePay() sur l'objet Staff pour calculer la rémunération de ce personnel.

Enfin, nous utilisons la méthode ToString() pour obtenir des informations sur l'objet Staff et afficher ces informations à l'écran à l'aide de la méthode Console.WriteLine().

Essayez de coder ce bloc d'essai vous-même.

Ensuite, dans le bloc catch, nous essayons de détecter toutes les erreurs qui pourraient se produire. Dans ce bloc catch, nous affichons simplement un message d'erreur et réduisons la valeur de i de un (i-- ;) de sorte que la boucle for soit répétée pour l'objet Staff actuel au lieu de passer à l'élément suivant dans myStaff.

Essayez de coder vous-même ce bloc de capture.

Avec cela, nous sommes arrivés à la fin de la boucle for. Nous sommes maintenant prêts à générer les fiches de paie pour chaque membre du personnel. Pour ce faire, nous devons d'abord déclarer et instancier un objet PaySlip. Appelons cet objet ps. Nous passons les variables month et year au constructeur lors de l'instanciation de l'objet.

Ensuite, nous utilisons l'objet ps pour appeler les fonctions GeneratePaySlip() et GenerateSummary () et passer dans myStaff comme argument. Enfin, nous ajoutons une instruction Console.Read() ; pour éviter que la console ne se ferme immédiatement après la fin du programme.

C'est fait ?

Si vous avez réussi à coder le programme Main(), donnez-vous une tape sur les épaules. Vous venez de coder un programme complet en C# ! Bien joué !

Si vous avez des problèmes pour le coder, continuez d'essayer. Vous pouvez vous référer à la solution suggérée dans l'annexe A pour référence.

Une fois que vous avez fini de coder la méthode Main()-, vous êtes prêt à exécuter votre programme. Vous êtes excité ? Allons-y !

Cliquez sur le bouton "Démarrer" pour lancer le programme et entrez les valeurs demandées.

Les fiches de paie générées se trouvent dans le même dossier que le fichier .exe, qui se trouve dans le dossier CSProject > CSProject > Bin > Debug. Essayez de faire des erreurs et de saisir des lettres alphabétiques au lieu de chiffres. Jouez avec le programme pour voir comment il fonctionne. Est-ce que tout fonctionne comme prévu ? Si c'est le cas, c'est parfait ! Vous avez fait un excellent travail ! Essayez de trouver des moyens d'améliorer le logiciel. Par exemple, vous pouvez inclure davantage de contrôles pour vous assurer que les utilisateurs ont saisi les valeurs correctes pour l'année et les heures travaillées.

Si votre code ne fonctionne pas, comparez-le avec l'exemple de réponse et essayez de comprendre ce qui n'a pas fonctionné. Vous apprendrez beaucoup en analysant vos erreurs. C'est en résolvant des problèmes que l'on s'amuse et que l'on obtient la plus grande récompense. Amusez-vous et n'abandonnez jamais ! Vous trouverez un exemple de réponse à l'annexe A.

**Merci**

Nous sommes arrivés à la fin du livre. Merci d'avoir lu ce livre et j'espère que vous l'avez apprécié. Plus important encore, j'espère sincèrement que le livre vous a aidé à maîtriser les bases de la programmation C#.

Je sais que vous auriez pu choisir parmi une douzaine de livres sur la programmation C#, mais vous avez pris un risque avec ce livre. Merci encore une fois d'avoir téléchargé ce livre et de l'avoir lu jusqu'au bout. Veuillez essayer les exercices et le projet.

Vous apprendrez beaucoup en le faisant.

J'aimerais maintenant vous demander une "petite" faveur. Pourriez-vous prendre une minute ou deux pour laisser une critique de ce livre sur Amazon ?

Cette critique m'aidera énormément et me permettra de continuer à écrire d'autres guides sur la programmation. Si vous aimez le livre ou si vous avez des suggestions d'amélioration, faites-le moi savoir. Je vous en serai profondément reconnaissant :)

Enfin, n'oubliez pas que vous pouvez télécharger le code source du projet sur http://www.learncodingfast.com/csharp.

Vous pouvez également me contacter à jamie@learncodingfast.com.