# Module 4: Création de classes et implémentation de collections de type sécurisé

# **Contenu:**

|  |  |
| --- | --- |
| **Leçon 1:** | [Créer des classes](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C04.html#P1) |
| **Leçon 2:** | [Définition et mise en œuvre d'interfaces](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C04.html#P2) |
| **Lecon 3:** | [Implémentation de collections de type sécurisé](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C04.html#P3) |
| **Laboratoire:** | [Ajout de la validation des données et de la sécurité de type à l'application](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C04.html#P4) |
|  | [Revue du module et points à retenir](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C04.html#P5) |

# **Présentation du module**

Les classes vous permettent de créer vos propres types personnalisés, autonomes et réutilisables. Les interfaces vous permettent de définir un ensemble d'entrées et de sorties que les classes doivent implémenter afin d'assurer la compatibilité avec les consommateurs des classes. Dans ce module, vous apprendrez à utiliser des interfaces et des classes pour définir et créer vos propres types personnalisés et réutilisables. Vous apprendrez également à créer et à utiliser des collections énumérables et de type sécurisé de tout type.

### **Objectifs**

Après avoir terminé ce module, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Créez et instanciez des classes. |
| • | Créer et instancier des interfaces |
| • | Utiliser des génériques pour créer des collections de type sécurisé |

# Leçon 1: Créer des classes

Dans Visual C #, vous pouvez définir vos propres types personnalisés en créant des classes. En tant que construction de programmation, la classe est au cœur de la programmation orientée objet dans Visual C #. Il vous permet d'encapsuler les comportements et les caractéristiques de toute entité logique de manière réutilisable et extensible.

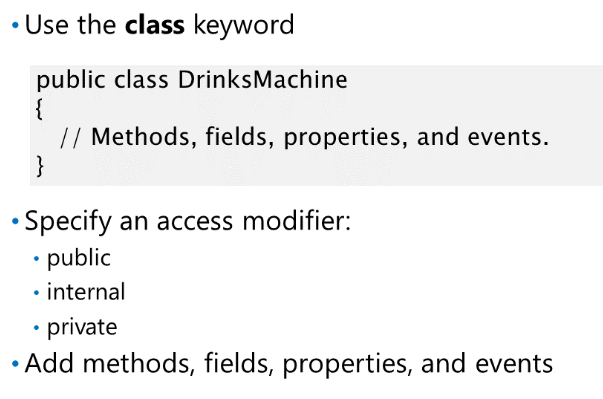
Dans cette leçon, vous apprendrez à créer, utiliser et tester des classes dans vos propres applications.

## ****Objectifs de la leçon****

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Créez des classes. |
| • | Créez des objets en instanciant des classes. |
| • | Utilisez des constructeurs pour définir des valeurs ou pour exécuter la logique lorsque les classes sont instanciées. |
| • | Expliquez la différence entre les types de référence et les types de valeur. |
| • | Créez des classes et des membres statiques. |
| • | Décrivez le processus de haut niveau pour tester la fonctionnalité de classe. |

## ****Création de classes et de membres****



Dans Visual C #, une classe est une construction de programmation que vous pouvez utiliser pour définir vos propres types personnalisés. Lorsque vous créez une classe, vous créez un nouveau type, qui crée effectivement un plan directeur pour l'instance. La classe définit les comportements et les caractéristiques, ou membres de classe, qui existent dans toutes les instances de la classe. Vous représentez ces comportements et caractéristiques en définissant des méthodes, des champs, des propriétés et des événements au sein de votre classe.

### **Déclarer une classe**

Par exemple, supposons que vous créez une classe nommée DrinksMachine.

Vous utilisez le mot clé class pour déclarer une classe, comme illustré dans l'exemple suivant: You use the class keyword to declare a class, as shown in the following example:

**Déclarer une classe**

class public DrinksMachine

{

// Les méthodes, les champs, les propriétés et les événements se trouvent ici.

}

Le mot-clé class est précédé d'un modificateur d'accès, tel que public dans l'exemple ci-dessus, qui spécifie où vous pouvez utiliser le type. Vous pouvez utiliser les modificateurs d'accès suivants dans vos déclarations de classe:

| **Modificateur d'accès** | **La description** |
| --- | --- |
| **public** | Le type est disponible pour le code exécuté dans n'importe quel assembly qui fait référence à l'assembly dans lequel la classe est contenue. |
| **internal** | Le type est disponible pour n'importe quel code dans le même assembly, mais pas disponible pour coder dans un autre assembly. Il s'agit de la valeur par défaut si vous ne spécifiez pas de modificateur d'accès. |
| **private** | Le type est uniquement disponible pour coder dans la classe qui le contient. Vous ne pouvez utiliser le modificateur d'accès privé qu'avec des classes imbriquées. |

**Ajouter des membres à une classe**

Vous utiliseriez des champs et des propriétés pour définir les caractéristiques d'une machine à boissons, telles que la marque, le modèle, l'âge et l'intervalle d'entretien de la machine. Vous créeriez des méthodes pour représenter les choses qu'une machine à boissons peut faire, comme préparer un expresso ou un cappuccino. Enfin, vous définiriez des événements pour représenter des actions qui pourraient nécessiter votre attention, telles que le remplacement des grains de café lorsque la machine est à court de grains de café.

Dans votre classe, vous pouvez ajouter des méthodes, des champs, des propriétés et des événements pour définir les comportements et les caractéristiques de votre type, comme illustré dans l'exemple suivant:

**Définition des membres de la classe**

Classe publique DrinksMachine

{

// Les instructions suivantes définissent une propriété avec un champ privé.

int \_age privé;

public int Âge

{

obtenir

{

retour \_ âge;

}

ensemble

{

if (valeur> 0)

\_age = valeur;

}

}

// Les instructions suivantes définissent les propriétés.

chaîne publique Make;

Modèle de chaîne publique;

// Les instructions suivantes définissent des méthodes.

public void MakeCappuccino ()

{

// La logique de la méthode va ici.

}

public void MakeEspresso ()

{

// La logique de la méthode va ici.

}

// L'instruction suivante définit un événement. La définition du délégué n'est pas affichée.

événement public OutOfBeansHandler OutOfBeans;

}

## **Instancier des classes**

Une classe n'est que la définition d'un type. Pour utiliser les comportements et les caractéristiques que vous définissez dans une classe, vous devez créer des instances de la classe.

Pour créer une nouvelle instance d'une classe, vous utilisez le nouveau mot-clé, comme illustré dans l'exemple suivant:

**Instancier une classe**

DrinksMachine dm = nouveau DrinksMachine ();

Lorsque vous instanciez une classe de cette manière, vous faites en fait deux choses:

|  |  |
| --- | --- |
| • | Vous créez un nouvel objet en mémoire basé sur le type DrinksMachine. |
| • | Vous créez une référence d'objet nommée dm qui fait référence au nouvel objet DrinksMachine. |

Lorsque vous créez votre référence d'objet, au lieu de spécifier explicitement le type DrinksMachine, vous pouvez autoriser le compilateur à déduire le type de l'objet au moment de la compilation. C'est ce qu'on appelle l'inférence de type. Pour utiliser l'inférence de type, vous créez votre référence d'objet à l'aide du mot clé var, comme illustré dans l'exemple suivant: To use type inference, you create your object reference by using the var keyword, as shown in the following example:

**Instanciation d'une classe à l'aide de l'inférence de type**

var dm = nouveau DrinksMachine ();

Dans ce cas, le compilateur ne connaît pas à l'avance le type de la variable dm. Lorsque la variable dm est initialisée en tant que référence à un objet DrinksMachine, le compilateur en déduit que le type de dm est DrinksMachine. L'utilisation de l'inférence de type de cette manière n'entraîne aucun changement dans la façon dont votre application s'exécute; c'est simplement un raccourci pour vous éviter de taper deux fois le nom de la classe. Dans certaines circonstances, l'inférence de type peut rendre votre code plus facile à lire, tandis que dans d'autres circonstances, elle peut rendre votre code plus confus. En règle générale, envisagez d'utiliser l'inférence de type lorsque le type de variable est absolument clair.

Après avoir instancié votre objet, vous pouvez utiliser l'un des membres (méthodes, champs, propriétés et événements) que vous avez définis dans la classe, comme illustré dans l'exemple suivant:

**Utilisation des membres d'objet**

var dm = nouveau DrinksMachine ();

dm.Make = "Quatrième café";

dm.Model = "Beancrusher 3000";

dm.Age = 2;

dm.MakeEspresso ();

Cette approche pour appeler des membres sur une variable d'instance est connue sous le nom de notation par points. Vous tapez le nom de la variable, suivi d'un point, suivi du nom du membre. La fonctionnalité IntelliSense de Visual Studio vous invite avec les noms des membres lorsque vous tapez un point après une variable.

## **Utilisation de constructeurs**

Dans les rubriques précédentes, vous avez peut-être remarqué que la syntaxe pour instancier une classe - par exemple, new DrinksMachine () - ressemble à la syntaxe pour appeler une méthode. En effet, lorsque vous instanciez une classe, vous appelez en fait une méthode spéciale appelée constructeur. Un constructeur est une méthode de la classe qui porte le même nom que la classe.

Les constructeurs sont souvent utilisés pour spécifier des valeurs initiales ou par défaut pour les membres de données dans le nouvel objet, comme le montre l'exemple suivant:

**Ajouter un constructeur**

Classe publique DrinksMachine

{

public int Age {get; ensemble; }

public DrinksMachine ()

{

Âge = 0;

}

}

Un constructeur qui n'accepte aucun paramètre est appelé constructeur par défaut. Ce constructeur est appelé chaque fois que quelqu'un instancie votre classe sans fournir d'arguments. Si vous n'incluez pas de constructeur dans votre classe, le compilateur Visual C # ajoutera automatiquement un constructeur par défaut public vide à votre classe compilée.

Dans de nombreux cas, il est utile pour les consommateurs de votre classe de pouvoir spécifier des valeurs initiales pour les membres de données lorsque la classe est instanciée. Par exemple, lorsqu'une personne crée une nouvelle instance de DrinksMachine, il peut être utile qu'elle puisse spécifier la marque et le modèle de la machine en même temps. Votre classe peut inclure plusieurs constructeurs avec différentes signatures qui permettent aux consommateurs de fournir différentes combinaisons d'informations lorsqu'ils instancient votre classe.

L'exemple suivant montre comment ajouter plusieurs constructeurs à une classe:

**Ajout de plusieurs constructeurs**

Classe publique DrinksMachine

{

public int Age {get; ensemble; }

chaîne publique Make {get; ensemble; }

chaîne publique Model {get; ensemble; }

Public DrinksMachine (âge int.)

{

this.Age = âge;

}

public DrinksMachine (marque de chaîne, modèle de chaîne)

{

this.Make = faire;

this.Model = modèle;

}

public DrinksMachine (âge int, marque de chaîne, modèle de chaîne)

{

this.Age = âge;

this.Make = faire;

this.Model = modèle;

}

}

Les consommateurs peuvent utiliser l'un des constructeurs pour créer des instances de votre classe, en fonction des informations dont ils disposent à ce moment-là. Par exemple:

**Appel aux constructeurs**

var dm1 = nouveau DrinksMachine (2);

var dm2 = new DrinksMachine ("Fourth Coffee", "BeanCrusher 3000");

var dm3 = new DrinksMachine (3, "Quatrième café", "BeanToaster Turbo");

## **Types de référence et types de valeur**

Maintenant que vous savez comment créer et instancier des classes, vous découvrirez les différences entre les classes et les structures.

Tout d'abord, voyons cependant comment Visual C # fonctionne dans les coulisses.

**Le Common Language Runtime (CLR)**

Le framework .Net se compose de plusieurs composants, construits les uns sur les autres. Le premier est le CLR. C'est le moteur qui enveloppe et exécute toutes les applications .Net. Le CLR est responsable - entre autres: de l'exécution du code .Net, de la gestion de la mémoire et du garbage collection pour l'application, de la gestion des exceptions, etc.

Au-dessus du CLR se trouve la bibliothèque de classes de base (BCL), qui comprend tous les types intégrés dans le .Net Framework.

Ci-dessus, l'implémentation spécifique des différents langages pris en charge par .Net.

**Lecture supplémentaire:**Pour plus d'informations sur le CLR, consultez:<https://aka.ms/moc-20483c-m4-pg3>.

**Le modèle de mémoire .Net**

Voyons maintenant comment les variables sont réellement enregistrées dans la mémoire de l'ordinateur. .Net utilise deux méthodes principales pour sauvegarder sa mémoire de travail.

**La pile**

La pile est utilisée pour enregistrer les variables de valeur locales dans la portée actuelle. Il fonctionne exactement comme la collection de piles, chaque nouvelle variable étant assignée juste à côté de la dernière utilisée. Lorsque le code quitte sa portée actuelle, toutes les données associées dans la pile sont effacées et la pile revient à sa portée précédente. Une portée dans .Net est généralement définie comme le bloc de code en cours d'exécution entouré par des accolades. L'utilisation la plus courante est la méthode. Lorsqu'une méthode est entrée, la pile enregistre toutes les variables de valeur locales créées dans cette méthode, ainsi que tous les paramètres de valeur passés à la méthode. Lorsque le code quitte la méthode, toutes ces données sont effacées. Cependant, si une autre méthode est appelée à partir de la méthode actuelle, les nouvelles données seront enregistrées juste après les données actuelles.

La pile a une taille fixe pendant la durée de vie de l'application. Le dépassement entraînera une exception StackOverflowException.

**Le tas**

Le tas est utilisé pour enregistrer de grandes variables de référence. La mémoire est allouée dynamiquement à partir du tas selon les besoins. Lorsqu'un nouvel objet volumineux est créé, le CLR recherche l'espace mémoire libre continu suivant suffisamment grand pour contenir l'objet et lui alloue cette mémoire.

La mémoire n'est pas automatiquement effacée du tas lorsque la portée est quittée comme elle l'était avec la pile. Au contraire, le garbage collector est chargé d'effacer le tas. Lorsqu'il le décide, le garbage collector lancera une analyse des objets dans le tas. Il continue de supprimer tout objet qu'il détermine n'est plus utilisé. Enfin, le garbage collector défragmente la mémoire utilisée pour éliminer les petits espaces entre les objets existants. Ceci permet d'éviter le gaspillage car les objets sur le tas ne sont alloués qu'à des espaces mémoire libres continus.

Le tas n'a pas de taille fixe et peut croître pendant la durée de vie de l'application. Cependant, il a une limite supérieure à sa taille. Si l'application atteint cette limite supérieure, elle lève une exception OutOfMemoryException.

**Définir les types de référence et de valeur**

Maintenant que nous savons où et comment le CLR stocke les variables de notre application, voyons ce qu'il enregistre, où et comment interagir avec elles

**Types de référence**

Ceux-ci sont supposés être des objets assez volumineux et ne sont donc désignés que par leur adresse (d'où le nom). Lorsque vous créez un objet de type référence, vous allouez toute la mémoire dont il a besoin, cependant, ce que vous recevez en retour n'est pas l'objet lui-même, mais uniquement son adresse dans la mémoire. Lorsque vous «copiez» un type de référence, comme indiqué ci-dessous:

Var x = new MyClass ();

Var y = x;

Seule la référence est copiée. Toute modification apportée aux données dans l'instance s'affiche lorsque x ou y est interrogé, car il s'agit en fait de la même instance.

x.MyProp = 42;

Console.WriteLine (y.MyProp); // Imprimer 42

y.MyProp + = 100;

Console.WriteLine (x.MyProp); // Imprimer 142

Cela signifie également que toute modification apportée à un type référencé passé en tant que paramètre à une méthode sera présente lorsque la méthode se termine. Les types de référence ne sont enregistrés que sur le tas.

Dans Visual C #, les classes et les délégués sont toujours des types de référence. Pour créer vous-même un nouveau type de référence, il vous suffit de définir une nouvelle classe. Cela s'applique également aux types intégrés, vous pouvez voir si un certain type est un type de référence en recherchant sa définition.

**Types de valeur**

Ceux-ci sont supposés être des objets plus petits et ne sont pas référencés indirectement. Lorsqu'une variable de type valeur est créée, cette variable conservera la valeur de la variable elle-même. Lorsque vous copiez un type valeur dans un autre, il est réellement copié, ce qui donne deux instances différentes avec la même valeur. Un changement au premier n'affectera pas l'autre.

Var x = 1;

Var y = x;

x = 42

Console.WriteLine (x); // Imprimer 42

Console.WriteLine (y); // Imprimer 1

y + = 100;

Console.WriteLine (x); // Imprimer 42

Console.WriteLine (y); // Imprimer 101

Lorsqu'un type de valeur est affecté en tant que paramètre de méthode, sa valeur est à nouveau copiée et la méthode utilise la valeur copiée. Cela signifie que toute modification de la valeur du paramètre ne sera pas vue lors du retour de la méthode.

Les types de valeur sont parfois enregistrés sur le tas, et parfois sur la pile, en fonction de leur utilisation. En règle générale, seuls les membres et paramètres de la méthode locale sont enregistrés sur la pile. Alors que d'autres utilisations (comme les champs de classe et les propriétés) sont enregistrées sur le tas en tant que partie de leur objet parent.

Dans Visual C #, seules les structures sont des types valeur. Pour créer vous-même un nouveau type de valeur, il vous suffit de définir une nouvelle structure. Cela s'applique également aux types intégrés, vous pouvez voir si un certain type est un type valeur en recherchant sa définition.

**Noter:**La plupart des types intégrés dans Visual C #, tels que int, bool, byte et char, sont des types valeur. Pour plus d'informations sur les types intégrés, consultez la page Table des types intégrés (référence C #) à l'adresse<http://go.microsoft.com/fwlink/?LinkID=267800>.

**Boxe et déballage**

Dans certains scénarios, vous devrez peut-être convertir des types valeur en types référence, et vice versa. Par exemple, certaines classes de collection n'accepteront que les types de référence. Cela risque moins de poser problème avec l'avènement des collections génériques. Cependant, vous devez toujours être conscient du concept, car un concept fondamental de Visual C # est que vous pouvez traiter n'importe quel type comme un objet.

Le processus de conversion d'un type valeur en type référence est appelé boxing. Pour encadrer une variable, vous l'affectez à une référence d'objet:

**Boxe**

int i = 100;

objet o = i;

Le processus de boxe est implicite. Lorsque vous affectez une référence d'objet à un type valeur, le compilateur Visual C # crée automatiquement un objet pour encapsuler la valeur et le stocke en mémoire. Si vous copiez la référence d'objet, la copie pointera vers le même wrapper d'objet en mémoire.

Le processus de conversion d'un type de référence en type valeur est appelé unboxing. Contrairement au processus de boxe, pour déballer un type de valeur, vous devez explicitement convertir la variable dans son type d'origine:

**Déballage**

int j;

j = (entier) o;

**Noter:**Notez que la boxe et le déballage sont un processus lent et coûteux. Il est généralement préférable de conserver le type spécifié si possible. L'utilisation de génériques est une bonne solution au problème. En savoir plus sur les génériques dans la leçon 3 - Implémentation de collections de type sécurisé.

## **Démonstration: comparaison des types de référence et des types de valeur**

Dans cette démonstration, vous allez créer un type valeur pour stocker une valeur entière et un type référence pour stocker une valeur entière. Vous allez créer une copie de chaque type, puis observer ce qui se passe lorsque vous modifiez la valeur de la copie dans chaque cas.

### **Démonstration étapes**

Vous trouverez les étapes dans la section «Démonstration: comparaison des types de référence et des types de valeur» sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md>.

## **Création de classes et de membres statiques**

Dans certains cas, vous souhaiterez peut-être créer une classe uniquement pour encapsuler des fonctionnalités utiles, plutôt que pour représenter une instance de quoi que ce soit. Par exemple, supposons que vous vouliez créer un ensemble de méthodes qui convertissent les poids et mesures impériaux en poids et mesures métriques, et vice versa. Cela n'aurait aucun sens si vous deviez instancier une classe pour utiliser ces méthodes, car vous n'avez pas besoin de stocker ou de récupérer des données spécifiques à l'instance. En fait, le concept d'instance n'a pas de sens dans ce cas.

Dans des scénarios comme celui-ci, vous pouvez créer une classe statique. Une classe statique est une classe qui ne peut pas être instanciée. Pour créer une classe statique, vous utilisez le mot clé static. Tous les membres de la classe doivent également utiliser le mot clé static, comme illustré dans l'exemple suivant:

**Classes statiques**

Conversions de classe statique publique

{

public static double PoundsToKilos (double livres)

{

// Convertit l'argument de livres en kilogrammes

kilos doubles = livres \* 0,4536;

renvoyer les kilos;

}

public static double KilosToPounds (double kilos)

{

// Convertit l'argument de kilogrammes en livres

livres doubles = kilos \* 2,205;

rendre les livres;

}

}

Pour appeler une méthode sur une classe statique, vous appelez la méthode sur le nom de classe lui-même plutôt que sur un nom d'instance, comme le montre l'exemple suivant:

**Appel de méthodes sur une classe statique**

double weightInKilos = 80;

double weightInPounds = Conversions.KilosToPounds (weightInKilos);

**Membres statiques**

Les classes non statiques peuvent inclure des membres statiques. Ceci est utile lorsque certains comportements et caractéristiques se rapportent à l'instance (membres de l'instance), tandis que certains comportements et caractéristiques se rapportent au type lui-même (membres statiques). Les méthodes, les champs, les propriétés et les événements peuvent tous être déclarés statiques. Les propriétés statiques sont souvent utilisées pour renvoyer des données communes à toutes les instances ou pour suivre le nombre d'instances d'une classe créées. Les méthodes statiques sont souvent utilisées pour fournir des utilitaires liés au type d'une certaine manière, tels que des fonctions de comparaison.

Pour déclarer un membre statique, vous utilisez le mot clé static avant le type de retour du membre, comme illustré par l'exemple suivant:

**Membres statiques dans les classes non statiques**

Classe publique DrinksMachine

{

public int Age {get; ensemble; }

chaîne publique Make {get; ensemble; }

chaîne publique Model {get; ensemble; }

public static int CountDrinksMachines ()

{

// Ajoutez la logique de méthode ici.

}

}

Quel que soit le nombre d'instances de votre classe, il n'y a qu'une seule instance d'un membre statique. Vous n'avez pas besoin d'instancier la classe pour utiliser des membres statiques. Vous accédez aux membres statiques via le nom de la classe plutôt que le nom de l'instance, comme le montre l'exemple suivant:

**Accéder aux membres statiques**

int drinksMachineCount = DrinksMachine.CountDrinksMachines ();

## **Cours de test**

Les classes représentent souvent des unités de fonctionnalité autonomes. Dans de nombreux cas, vous souhaiterez tester les fonctionnalités de vos classes de manière isolée avant de les intégrer à d'autres classes dans vos applications.

Pour tester les fonctionnalités de manière isolée, vous créez un test unitaire. Un test unitaire présente le code sous test avec des entrées connues, effectue une action sur le code sous test (par exemple en appelant une méthode), puis vérifie que les sorties de l'opération sont comme attendues. De cette façon, le test unitaire représente un contrat que votre code doit remplir. Cependant, lorsque vous modifiez l'implémentation d'une classe ou d'une méthode, le test unitaire garantit que votre code renvoie toujours des sorties particulières en réponse à des entrées particulières.

Par exemple, considérons le cas où vous créez une classe simple pour représenter un client. Pour vous aider à cibler vos efforts marketing, la classe Customer inclut une méthode GetAge qui renvoie l'âge actuel du client en années:

**Classe sous test**

Client de classe publique

{

public DateTime DateOfBirth {get; ensemble; }

public int GetAge ()

{

TimeSpan difference = DateTime.Now.Subtract (DateOfBirth);

int ageInYears = (int) (difference.Days / 365.25);

// Remarque: la conversion d'un double en un entier arrondit au nombre entier le plus proche.

return ageInYears;

}

}

Dans ce cas, vous souhaiterez peut-être créer un test unitaire qui garantit que la méthode GetAge se comporte comme prévu. En tant que tel, votre méthode de test doit instancier la classe Customer, spécifier une date de naissance, puis vérifier que la méthode GetAge renvoie l'âge correct en années. Selon le framework de test unitaire que vous utilisez, votre méthode de test peut ressembler à ceci:

**Exemple de méthode d'essai**

[Méthode d'essai]

public void TestGetAge ()

{

// Organiser.

DateTime dob = DateTime.Today;

dob.AddDays (7);

dob.AddYears (-24);

Client testCust = nouveau client ();

testCust.DateOfBirth = dob;

// Le 24e anniversaire du client est dans sept jours, donc l'âge en années doit être de 23 ans.

int attenduAge = 23;

// Agir.

int actualAge = testCust.GetAge ();

// Assert.

// Échec du test si l'âge réel et l'âge attendu sont différents.

Assert.IsTrue ((actualAge == expectedAge), "L'âge n'est pas calculé correctement");

}

Notez que la méthode de test unitaire est divisée en trois phases conceptuelles:

|  |  |
| --- | --- |
| • | **Organiser**. Dans cette phase, vous créez les conditions du test. Vous instanciez la classe que vous souhaitez tester et vous configurez toutes les valeurs d'entrée requises par le test. |
| • | **Acte**. Dans cette phase, vous effectuez l'action que vous souhaitez tester. |
| • | **Affirmer**. Dans cette phase, vous vérifiez les résultats de l'action. Si les résultats ne correspondent pas aux attentes, le test échoue. |

La méthode Assert.IsTrue fait partie du Microsoft Unit Test Framework qui est inclus dans Visual Studio 2012. Cette méthode particulière lève une exception si la condition spécifiée n'est pas évaluée à true. Cependant, les concepts décrits ici sont communs à tous les frameworks de tests unitaires.

# Leçon 2: Définition et implémentation d'interfaces

Une interface est un peu comme une classe sans implémentation. Il spécifie un ensemble de caractéristiques et de comportements en définissant des signatures pour les méthodes, les propriétés, les événements et les indexeurs, sans spécifier comment l'un de ces membres est implémenté. Lorsqu'une classe implémente une interface, la classe fournit une implémentation pour chaque membre de l'interface. En implémentant l'interface, la classe garantit ainsi qu'elle fournira la fonctionnalité spécifiée par l'interface.

Dans cette leçon, vous apprendrez à définir et à implémenter des interfaces.

## **Objectifs de la leçon**

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Expliquez le but des interfaces dans Visual C #. |
| • | Créez des interfaces. |
| • | Créez des classes qui implémentent une seule interface. |
| • | Créez des classes qui implémentent plusieurs interfaces. |
| • | Implémentez l'interface IComparable. |
| • | Implémentez l'interface IComparer. |

## **Présentation des interfaces**

Dans Visual C #, une interface spécifie un ensemble de caractéristiques et de comportements; il le fait en définissant des méthodes, des propriétés, des événements et des indexeurs. L'interface elle-même ne spécifie pas comment ces membres sont implémentés. Au lieu de cela, les classes peuvent implémenter l'interface et fournir leurs propres implémentations des membres de l'interface. Vous pouvez considérer une interface comme un contrat. En implémentant une interface particulière, une classe garantit aux consommateurs qu'elle fournira des fonctionnalités spécifiques via des membres spécifiques, même si l'implémentation réelle ne fait pas partie du contrat.

Par exemple, supposons que vous souhaitiez développer un système de cartes de fidélité pour Fourth Coffee. Vous pouvez commencer par créer une interface nommée ILoyaltyCardHolder qui définit:

|  |  |
| --- | --- |
| • | Une propriété entière en lecture seule nommée TotalPoints. |
| • | Une méthode nommée AddPoints qui accepte un argument décimal. |
| • | Une méthode nommée ResetPoints. |

L'exemple suivant montre une interface qui définit une propriété en lecture seule et deux méthodes:

**Définition d'une interface**

interface publique ILoyaltyCardHolder

{

int TotalPoints {obtenir; }

int AddPoints (valeur décimale de transaction);

void ResetPoints ();

}

**Noter:**La convention de programmation stipule que tous les noms d'interface doivent commencer par un "I".

Notez que les méthodes de l'interface n'incluent pas de corps de méthode. De même, les propriétés de l'interface indiquent les accesseurs à inclure mais ne fournissent aucun détail d'implémentation. L'interface indique simplement que toute classe d'implémentation doit inclure et fournir une implémentation pour les trois membres. Le créateur de la classe d'implémentation peut choisir comment les méthodes sont implémentées. Par exemple, toute implémentation de la méthode AddPoints acceptera un argument décimal (la valeur monétaire de la transaction client) et retournera un entier (le nombre de points ajoutés). Le développeur de classe peut implémenter cette méthode de différentes manières. Par exemple, une implémentation de la méthode AddPoints pourrait:

|  |  |
| --- | --- |
| • | Calculez le nombre de points à ajouter en multipliant la valeur de la transaction par un montant fixe. |
| • | Obtenez le nombre de points à ajouter en appelant un service. |
| • | Calculez le nombre de points à ajouter en utilisant des facteurs supplémentaires, tels que l'emplacement du titulaire de la carte de fidélité. |

L'exemple suivant montre une classe qui implémente l'interface ILoyaltyCardHolder:

**Implémentation d'une interface**

Client de classe publique: ILoyaltyCardHolder

{

private int totalPoints;

public int TotalPoints

{

get {return totalPoints; }

}

public int AddPoints (valeur de transaction décimale)

{

int points = Decimal.ToInt32 (transactionValue);

totalPoints + = points;

}

public void ResetPoints ()

{

totalPoints = 0;

}

// Autres membres de la classe Customer.

}

Les détails de l'implémentation n'ont pas d'importance pour l'appel des classes. En implémentant l'interface ILoyaltyCardHolder, la classe d'implémentation indique aux consommateurs qu'elle s'occupera de l'opération AddPoints. L'un des principaux avantages des interfaces est qu'elles vous permettent de modulariser votre code. Vous pouvez modifier la manière dont votre classe implémente l'interface à tout moment, sans avoir à mettre à jour les classes de consommateurs qui reposent sur une implémentation d'interface.

## **Définition des interfaces**

La syntaxe de définition d'une interface est similaire à la syntaxe de définition d'une classe.

Vous utilisez le mot clé interface pour déclarer une interface, comme le montre l'exemple suivant:

**Déclaration d'une interface**

interface publique IBeverage

{

// Les méthodes, propriétés, événements et indexeurs vont ici.

}

Semblable à une déclaration de classe, une déclaration d'interface peut inclure un modificateur d'accès. Vous pouvez utiliser les modificateurs d'accès suivants dans vos déclarations d'interface:

| **Modificateur d'accès** | **La description** |
| --- | --- |
| **Publique** | L'interface est disponible pour le code exécuté dans n'importe quel assembly. |
| **interne** | L'interface est disponible pour n'importe quel code dans le même assembly, mais n'est pas disponible pour coder dans un autre assembly. Il s'agit de la valeur par défaut si vous ne spécifiez pas de modificateur d'accès. |

**Ajout de membres d'interface**

Une interface définit la signature des membres mais n'inclut aucun détail d'implémentation. Les interfaces peuvent inclure des méthodes, des propriétés, des événements et des indexeurs:

|  |  |
| --- | --- |
| • | Pour définir une méthode, vous spécifiez le nom de la méthode, le type de retour et tous les paramètres:  int GetServingTemperature (bool includesMilk); |
| • | Pour définir une propriété, vous spécifiez le nom de la propriété, le type de la propriété et les accesseurs de propriété:  booléen IsFairTrade {obtenir; ensemble; } |
| • | Pour définir un événement, vous utilisez le mot-clé event, suivi du délégué du gestionnaire d'événements, suivi du nom de l'événement:  event EventHandler OnSoldOut; |
| • | Pour définir un indexeur, vous spécifiez le type de retour et les accesseurs: |

string this [index int] {get; ensemble; }

Les membres de l'interface n'incluent pas de modificateurs d'accès. Le but de l'interface est de définir les membres qu'une classe d'implémentation doit exposer aux consommateurs, afin que tous les membres de l'interface soient publics. Les interfaces ne peuvent pas inclure de membres liés à la fonctionnalité interne d'une classe, tels que des champs, des constantes, des opérateurs et des constructeurs.

## **Implémentation d'interfaces**

Pour créer une classe qui implémente une interface, vous ajoutez un signe deux-points suivi du nom de l'interface à votre déclaration de classe.

L'exemple suivant montre comment créer une classe qui implémente l'interface IBeverage:

**Déclaration d'une classe qui implémente une interface**

Café en classe publique: IBeverage

{

}

Au sein de votre classe, vous devez fournir une implémentation pour chaque membre de l'interface. Votre classe peut inclure des membres supplémentaires qui ne sont pas définis par l'interface. En fait, la plupart des classes incluront des membres supplémentaires, car généralement la classe étend l'interface. Cependant, vous ne pouvez omettre aucun membre d'interface de la classe d'implémentation. La façon dont vous implémentez les membres de l'interface n'a pas d'importance, tant que vos implémentations ont les mêmes signatures (c'est-à-dire les mêmes noms, types, types de retour et paramètres) que les définitions de membres dans l'interface.

L'exemple suivant montre une interface triviale, ainsi qu'une classe qui implémente l'interface:

**Implémentation d'une interface**

interface publique IBeverage

{

int GetServingTemperature (bool includesMilk);

booléen IsFairTrade {obtenir; ensemble; }

}

Café en classe publique: IBeverage

{

private int servantTempWithoutMilk {get; ensemble; }

private int servantTempWithMilk {get; ensemble; }

public int GetServingTemperature (bool includesMilk)

{

if (includesMilk)

{

return serveTempWithMilk;

}

autre

{

return serveTempWithoutMilk;

}

}

public bool IsFairTrade {obtenir; ensemble; }

// Les autres membres non-interface vont ici.

}

**Polymorphisme d'interface**

En ce qui concerne les interfaces, le polymorphisme indique que vous pouvez représenter une instance d'une classe comme une instance de toute interface implémentée par la classe. Le polymorphisme d'interface peut aider à augmenter la flexibilité et la modularité de votre code. Supposons que vous ayez plusieurs classes qui implémentent l'interface IBeverage, telles que Coffee, Tea, Juice, etc. Vous pouvez écrire du code qui fonctionne avec l'une de ces classes en tant qu'instances de IBeverage, sans connaître les détails de la classe d'implémentation. Par exemple, vous pouvez créer une collection d'instances IBeverage sans avoir besoin de connaître les détails de chaque classe qui implémente IBeverage.

Par exemple, si la classe Coffee implémente l'interface IBeverage, vous pouvez représenter un nouvel objet Coffee comme une instance de Coffee ou une instance de IBeverage:

**Représentation d'un objet en tant que type d'interface**

Café café1 = nouveau café ();

IBeverage coffee2 = nouveau café ();

Vous pouvez utiliser un cast implicite pour convertir en un type d'interface, car vous savez que la classe doit inclure tous les membres de l'interface.

**Diffusion vers un type d'interface**

Boisson alcoolisée = café1;

Vous devez utiliser un cast explicite pour convertir un type d'interface en un type de classe dérivé, car la classe peut inclure des membres qui ne sont pas définis dans l'interface.

**Conversion d'un type d'interface en type de classe dérivé**

Café coffee3 = boisson sous forme de café;

// OU ALORS

Café coffee4 = boisson (café);

## **Implémentation de plusieurs interfaces**

Dans de nombreux cas, vous souhaiterez créer des classes qui implémentent plusieurs interfaces. Par exemple, vous souhaiterez peut-être:

|  |  |
| --- | --- |
| • | Implémentez l'interface IDisposable pour permettre au runtime .NET de supprimer correctement votre classe. |
| • | Implémentez l'interface IComparable pour permettre aux classes de collection de trier les instances de votre classe. |
| • | Implémentez votre propre interface personnalisée pour définir les fonctionnalités de votre classe. |

Pour implémenter plusieurs interfaces, vous ajoutez une liste séparée par des virgules des interfaces que vous souhaitez implémenter dans votre déclaration de classe. Votre classe doit implémenter chaque membre de chaque interface que vous ajoutez à votre déclaration de classe.

L'exemple suivant montre comment créer une classe qui implémente plusieurs interfaces:

**Déclaration d'une classe qui implémente plusieurs interfaces**

Café de classe publique: IBeverage, IInventoryItem

{

}

**Implémentation implicite et explicite**

Lorsque vous créez une classe qui implémente une interface, vous pouvez choisir d'implémenter l'interface implicitement ou explicitement. Pour implémenter une interface implicitement, vous implémentez chaque membre d'interface avec une signature qui correspond à la définition de membre dans l'interface. Pour implémenter une interface explicitement, vous qualifiez entièrement chaque nom de membre afin qu'il soit clair que le membre appartient à une interface particulière.

L'exemple suivant montre une implémentation explicite de l'interface IBeverage:

**Implémentation explicite d'une interface**

Café en classe publique: IBeverage

{

private int servantTempWithoutMilk {get; ensemble; }

private int servantTempWithMilk {get; ensemble; }

public int IBeverage.GetServingTemperature (bool includesMilk)

{

if (includesMilk)

{

return serveTempWithMilk;

}

autre

{

return serveTempWithoutMilk;

}

}

public bool IBeverage.IsFairTrade {obtenir; ensemble; }

// Autres membres non-interface.

}

Certains développeurs préfèrent une implémentation d'interface explicite, car cela peut rendre le code plus facile à comprendre. Le seul scénario dans lequel vous devez utiliser l'implémentation d'interface explicite est si vous implémentez deux interfaces qui partagent un nom de membre. Par exemple, si vous implémentez des interfaces nommées IBeverage et IInventoryItem et que les deux interfaces déclarent une propriété booléenne nommée IsAvailable, vous devez implémenter au moins l'un des membres IsAvailable de manière explicite. Dans ce scénario, le compilateur serait incapable de résoudre la référence IsAvailable sans une implémentation explicite.

Il y a cependant une différence dans l'utilisation de la classe. Lors de l'implémentation implicite d'une interface, les membres peuvent être utilisés comme des membres publics normaux de la classe et peuvent être référencés sans aucune désignation spéciale. Cependant, lorsqu'une interface est implémentée explicitement, les membres ne sont rendus disponibles qu'en convertissant l'instance dans l'interface correspondante. Sinon, ils resteront cachés et le code les appelant ne se compilera pas.

L'exemple suivant montre l'utilisation d'une implémentation explicite de l'interface IBeverage. Supposons que la classe Coffee est la même que l'exemple précédent.

**Consommer une implémentation d'interface explicite**

Café myCoffee = nouveau café ();

// Cela ne compilera pas

myCoffee.GetServingTemperature (vrai)

IBeverage myBeverage = (IBeverage) myCoffee;

// Cela fonctionnera comme prévu

myBeverage.GetServingTemperature (vrai)

## **Implémentation de l'interface IComparable**

Le .NET Framework comprend diverses classes de collection qui vous permettent de trier le contenu de la collection. Ces classes, telles que la classe ArrayList, incluent une méthode nommée Sort. Lorsque vous appelez cette méthode sur une instance ArrayList, la collection ordonne son contenu.

Comment l'instance ArrayList sait-elle comment les éléments de la collection doivent être classés? Dans le cas des types simples, tels que les entiers, cela semble assez simple. Intuitivement, trois suit deux et deux suit un. Cependant, supposons que vous créez une collection d'objets Coffee. Comment l'instance ArrayList déterminerait-elle si un café est supérieur ou inférieur à un autre café? La réponse est que la classe Coffee doit fournir à l'instance ArrayList une logique qui lui permet de comparer un café avec un autre. Pour ce faire, la classe Coffee doit implémenter l'interface IComparable.

L'exemple suivant montre l'interface IComparable:

**L'interface IComparable**

interface publique IComparable

{

int CompareTo (Object obj);

}

Comme vous pouvez le voir, l'interface IComparable déclare une seule méthode nommée CompareTo. Les implémentations de cette méthode doivent:

|  |  |
| --- | --- |
| • | Comparez l'instance d'objet actuelle avec un autre objet du même type (l'argument). |
| • | Renvoie une valeur entière qui indique si l'instance d'objet actuelle doit être placée avant, à la même position ou après l'instance d'objet transmise. |

Les valeurs entières renvoyées par la méthode CompareTo sont interprétées comme suit:

|  |  |
| --- | --- |
| • | Inférieur à zéro indique que l'instance d'objet actuelle précède l'instance fournie dans l'ordre de tri. |
| • | Zéro indique que l'instance d'objet actuelle se trouve à la même position que l'instance fournie dans l'ordre de tri. |
| • | Plus de zéro indique que l'instance d'objet actuelle suit l'instance fournie dans l'ordre de tri. |

L'exemple suivant illustre ce qui se passe si vous utilisez la méthode CompareTo pour comparer deux entiers:

**Exemple CompareTo**

int nombre1 = 5;

int nombre2 = 100;

int result = nombre1.CompareTo (nombre2);

// La valeur du résultat est -1, indiquant que nombre1 doit précéder nombre2 dans l'ordre de tri.

**Noter:**Tous les types de valeur intégrés dans le .NET Framework implémentent l'interface IComparable. Pour plus d'informations sur l'interface IComparable, consultez la page IComparable Interface à l'adresse<https://aka.ms/moc-20483c-m4-pg1>.

Lorsque vous appelez la méthode Sort sur une instance ArrayList, la méthode Sort appelle la méthode CompareTo des membres de la collection pour déterminer l'ordre correct de la collection.

Lorsque vous implémentez l'interface IComparable dans vos propres classes, vous déterminez les critères selon lesquels les objets doivent être comparés. Par exemple, vous pouvez décider que les cafés doivent être triés par ordre alphabétique par variété.

L'exemple suivant montre comment implémenter l'interface IComparable:

**Implémentation de l'interface IComparable**

Café de classe publique: IComparable

{

public double AverageRating {get; ensemble; }

chaîne publique Variety {get; ensemble; }

int IComparable.CompareTo (objet obj)

{

Café coffee2 = obj as Coffee;

return String.Compare (this.Variety, coffee2.Variety);

}

}

Dans cet exemple, comme les valeurs que nous voulons comparer sont des chaînes, nous pouvons utiliser la méthode String.Compare. Cette méthode renvoie -1 si la chaîne actuelle précède la chaîne fournie dans un ordre de tri alphabétique, 0 si les chaînes sont identiques et 1 si la chaîne actuelle suit la chaîne fournie dans un ordre de tri alphabétique.

## **Implémentation de l'interface IComparer**

Lorsque vous appelez la méthode Sort sur une instance ArrayList, ArrayList trie la collection en fonction de l'implémentation de l'interface IComparable dans le type sous-jacent. Par exemple, si vous triez une collection ArrayList d'entiers, les critères de tri sont définis par l'implémentation de l'interface IComparable dans le type Int32. Le créateur de l'instance ArrayList n'a aucun contrôle sur les critères utilisés pour trier la collection.

Dans certains cas, les développeurs peuvent souhaiter trier les instances de votre classe à l'aide de critères de tri alternatifs. Par exemple, supposons que vous souhaitiez trier une collection d'instances Coffee par la valeur de la propriété AverageRating plutôt que par la propriété Variety. Pour trier une instance ArrayList à l'aide de critères de tri personnalisés, vous devez faire deux choses:

|  |  |
| --- | --- |
| 1. | Créez une classe qui implémente l'interface IComparer pour fournir votre fonctionnalité de tri personnalisé. |
| 2. | Appelez la méthode Sort sur l'instance ArrayList et transmettez une instance de votre implémentation IComparer en tant que paramètre. |

L'exemple suivant montre l'interface IComparer:

**L'interface IComparer**

interface publique IComparer

{

int Compare (Objet x, Objet y)

}

Comme vous pouvez le voir, l'interface IComparer déclare une seule méthode nommée Compare. Les implémentations de cette méthode doivent:

|  |  |
| --- | --- |
| • | Comparez deux objets du même type. |
| • | Renvoie une valeur entière qui indique si l'instance d'objet actuelle doit être placée avant, à la même position ou après l'instance d'objet transmise. |

L'exemple suivant montre comment implémenter l'interface IComparer:

**Implémentation de l'interface IComparer**

public class CoffeeRatingComparer: IComparer

{

public int Compare (Objet x, Objet y)

{

Café café1 = x comme café;

Café café2 = y comme café;

double rating1 = coffee1.AverageRating;

double rating2 = café2.AverageRating;

retournez rating1.CompareTo (rating2);

}

}

Dans l'exemple ci-dessus, comme les valeurs que nous voulons comparer sont des doubles, nous pouvons utiliser la méthode Double.CompareTo. Cela renvoie -1 si le double courant est inférieur au double fourni, 0 si le double courant est égal au double fourni et 1 si le double courant est supérieur au double fourni. Il est toujours préférable d'utiliser une fonction de comparaison intégrée, s'il en existe une, plutôt que de créer la vôtre.

L'exemple suivant montre comment utiliser une implémentation IComparer personnalisée:

**Utilisation d'une implémentation IComparer**

// Créez des instances de la classe Coffee.

Café café1 = nouveau café ();

café1.Évaluation = 4,5;

Café café2 = nouveau café ();

café2.Évaluation = 8,1;

Café café3 = nouveau café ();

café3.Évaluation = 7,1;

// Ajoute les instances Coffee à une ArrayList.

ArrayList coffeeList = new ArrayList ();

coffeeList.Add (café1);

coffeeList.Add (café2);

coffeeList.Add (café3);

// Trier la ArrayList par note moyenne.

coffeeList.Sort (nouveau CoffeeRatingComparer ());

Pour trier ArrayList à l'aide d'un comparateur personnalisé, vous appelez la méthode Sort et transmettez une nouvelle instance de votre implémentation IComparer en tant qu'argument.

# Leçon 3: Implémentation de collections de type sécurisé

Presque toutes les applications que vous créez utilisent des classes de collection sous une forme ou une autre. Dans la plupart des cas, les collections contiennent un ensemble d'objets du même type. Lorsque vous interagissez avec une collection, vous comptez souvent sur la collection pour fournir des objets d'un type spécifique. Historiquement, cela a créé divers défis. Vous deviez créer une logique de gestion des exceptions au cas où une collection contiendrait des éléments du type incorrect. Vous deviez également encadrer les types de valeur afin de les ajouter aux classes de collection et de les déballer lors de la récupération. Visual C # supprime plusieurs de ces défis en utilisant des génériques

Dans cette leçon, vous apprendrez à créer et à utiliser des classes génériques pour créer des collections fortement typées de tout type.

## **Objectifs de la leçon**

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Décrivez les génériques. |
| • | Identifiez les avantages des classes génériques par rapport aux classes non génériques. |
| • | Appliquez des contraintes aux paramètres de type. |
| • | Utilisez des collections de listes génériques. |
| • | Utilisez des collections de dictionnaires génériques. |
| • | Créez des collections génériques personnalisées. |
| • | Créez des collections génériques énumérables. |

## **Présentation des génériques**

Les génériques vous permettent de créer et d'utiliser des collections fortement typées qui sont de type sécurisé, ne nécessitent pas de cast d'éléments et ne nécessitent pas de boîte et de décompression de types de valeur. Les classes génériques fonctionnent en incluant un paramètre de type, T, dans la déclaration de classe ou d'interface. Vous n'avez pas besoin de spécifier le type de T tant que vous n'avez pas instancié la classe. Pour créer une classe générique, vous devez:

|  |  |
| --- | --- |
| • | Ajoutez le paramètre de type T entre crochets après le nom de la classe. |
| • | Utilisez le paramètre de type T à la place des noms de type dans vos membres de classe. |

L'exemple suivant montre comment créer une classe générique:

**Créer une classe générique**

classe publique CustomList <T>

{

public T this [index int] {get; ensemble; }

public void Add (élément T)

{

// La logique de la méthode va ici.

}

public void Supprimer (élément T)

{

// La logique de la méthode va ici.

}

}

Lorsque vous créez une instance de votre classe générique, vous spécifiez le type que vous souhaitez fournir en tant que paramètre de type. Par exemple, si vous souhaitez utiliser votre liste personnalisée pour stocker des objets de type Coffee, vous devez fournir Coffee comme paramètre de type.

L'exemple suivant montre comment instancier une classe générique:

**Instancier une classe générique**

CustomList <Coffee> clc = new CustomList <Coffee>;

Café café1 = nouveau café ();

Café café2 = nouveau café ();

clc.Add (café1);

clc.Add (café2);

Café d'abordCafé = clc [0];

Lorsque vous instanciez une classe, chaque instance de T dans la classe est effectivement remplacée par le paramètre de type que vous fournissez. Par exemple, si vous instanciez la classe CustomList avec un paramètre de type Coffee:

|  |  |
| --- | --- |
| • | La méthode Add n'acceptera qu'un argument de type Coffee. |
| • | La méthode Remove n'acceptera qu'un argument de type Coffee. |
| • | L'indexeur fournira toujours une valeur de retour de type Coffee. |

## **Avantages des génériques**

L'utilisation de classes génériques, en particulier pour les collections, offre trois avantages principaux par rapport aux approches non génériques: la sécurité de type, pas de casting, et pas de boxe et de déballage.

**Sécurité de type**

Prenons un exemple où vous utilisez une ArrayList pour stocker une collection d'objets Coffee. Vous pouvez ajouter des objets de n'importe quel type à une ArrayList. Supposons qu'un développeur ajoute un objet de type Tea à la collection. Le code se construira sans plainte. Cependant, une exception d'exécution se produira si la méthode Sort est appelée, car la collection ne peut pas comparer des objets de types différents. En outre, lorsque vous récupérez un objet de la collection, vous devez convertir l'objet dans le type correct. Si vous essayez de convertir l'objet en un type incorrect, une exception d'exécution de conversion non valide se produit.

L'exemple suivant montre les limitations de sécurité de type de l'approche ArrayList: The following example shows the type safety limitations of the ArrayList approach:

**Limitations de sécurité de type pour les collections non génériques**

var coffee1 = nouveau café ();

var coffee2 = nouveau café ();

var tea1 = nouveau thé ();

var arrayList1 = new ArrayList ();

arrayList1.Add (café1);

arrayList1.Add (café2);

arrayList1.Add (thé1);

// La méthode Sort lève une exception d'exécution car la collection n'est pas homogène.

arrayList1.Sort ();

// Le cast lève une exception d'exécution car vous ne pouvez pas convertir une instance de Tea en une instance de Coffee.

Café coffee3 = (Café) arrayList1 [2];

Comme alternative à ArrayList, supposons que vous utilisiez un List <T> générique pour stocker une collection d'objets Coffee. Lorsque vous instanciez la liste, vous fournissez un argument de type Coffee. Dans ce cas, votre liste est garantie d'être homogène, car votre code ne sera pas généré si vous essayez d'ajouter un objet d'un autre type. La méthode de tri fonctionnera car votre collection est homogène. Enfin, l'indexeur renvoie des objets de type Coffee, plutôt que System.Object, il n'y a donc aucun risque d'exceptions de cast non valides.

L'exemple suivant montre une alternative à l'approche ArrayList à l'aide de la classe générique List <T>:

**Sécurité des types dans les collections génériques**

var coffee1 = nouveau café ();

var coffee2 = nouveau café ();

var tea1 = nouveau thé ();

var genericList1 = nouvelle liste <Café> ();

genericList1.Add (café1);

genericList1.Add (café2);

// Cette ligne provoque une erreur de construction, car l'argument n'est pas de type Coffee.

genericList1.Add (thé1);

// La méthode Sort fonctionnera car la collection est garantie d'être homogène.

arrayList1.Sort ();

// L'indexeur renvoie des objets de type Coffee, il n'est donc pas nécessaire de convertir la valeur de retour.

Café coffee3 = genericList [1];

**Pas de casting**

Le moulage est un processus coûteux en calcul. Lorsque vous ajoutez des éléments à une ArrayList, vos éléments sont implicitement convertis en type System.Object. Lorsque vous récupérez des éléments à partir d'une ArrayList, vous devez les renvoyer explicitement à leur type d'origine. L'utilisation de génériques pour ajouter et récupérer des éléments sans conversion améliore les performances de votre application.

**Pas de boxe et de déballage**

Si vous souhaitez stocker des types de valeur dans une ArrayList, les éléments doivent être encadrés lorsqu'ils sont ajoutés à la collection et déballés lorsqu'ils sont récupérés. Le boxing et le déballage entraînent un coût de calcul important et peuvent ralentir considérablement vos applications, en particulier lorsque vous itérez sur de grandes collections. En revanche, vous pouvez ajouter des types de valeur à des listes génériques sans encadrer ni déballer la valeur.

L'exemple suivant montre la différence entre les collections génériques et non génériques en ce qui concerne la boxe et le déballage:

**Boxe et déballage: collections génériques et non génériques**

int nombre1 = 1;

var arrayList1 = new ArrayList ();

// Cette instruction encadre la valeur Int32 en tant que System.Object.

arrayList1.Add (nombre1);

// Cette instruction déballe la valeur Int32.

int nombre2 = (int) arrayList1 [0];

var genericList1 = nouvelle liste <Int32> ();

// Cette instruction ajoute une valeur Int32 sans boxing.

genericList1.Add (nombre1);

// Cette instruction récupère la valeur Int32 sans déballage.

int nombre3 = genericList1 [0];

## **Génériques contraignants**

Dans certains cas, vous devrez peut-être restreindre les types que les développeurs peuvent fournir comme arguments lorsqu'ils instancient votre classe générique. La nature de ces contraintes dépendra de la logique que vous implémentez dans votre classe générique. Par exemple, si une classe de collection utilise une propriété nommée AverageRating pour trier les éléments d'une collection, vous devez contraindre le paramètre type aux classes qui incluent la propriété AverageRating. Supposons que la propriété AverageRating est définie par l'interface IBeverage. Pour implémenter cette restriction, vous devez contraindre le paramètre type aux classes qui implémentent l'interface IBeverage à l'aide du mot clé where.

L'exemple suivant montre comment contraindre un paramètre de type à des classes qui implémentent une interface particulière:

**Contraindre les paramètres de type par interface**

public class CustomList <T> où T: IBeverage

{

}

Vous pouvez appliquer les six types de contraintes suivants aux paramètres de type:

| **Contrainte** | **La description** |
| --- | --- |
| **où T: <nom de l'interface>** | L'argument type doit être, ou implémenter, l'interface spécifiée. |
| **où T: <nom de la classe de base>** | L'argument de type doit être ou dériver de la classe spécifiée. |
| **où T: U** | L'argument de type doit être ou dériver de l'argument de type fourni U. |
| **où T: nouveau ()** | L'argument type doit avoir un constructeur par défaut public. |
| **où T: struct** | L'argument type doit être un type valeur. |
| **où T: classe** | L'argument type doit être un type référence. |

Vous pouvez également appliquer plusieurs contraintes à la même classe, comme le montre l'exemple suivant:

**Appliquer plusieurs contraintes de type**

classe publique CustomList <T> où T: IBeverage, IComparable <T>, new ()

{

}

## **Utilisation des collections de listes génériques**

L'une des utilisations les plus courantes et les plus importantes des génériques est dans les classes de collection. Les collections génériques se divisent en deux grandes catégories: les collections de listes génériques et les collections de dictionnaires génériques. Une liste générique stocke une collection d'objets de type T.

**La classe List <T>**

La classe List <T> fournit une alternative fortement typée à la classe ArrayList. Comme la classe ArrayList, la classe List <T> inclut des méthodes pour:

|  |  |
| --- | --- |
| • | Ajoutez un article. |
| • | Supprimer un élément. |
| • | Insérez un élément à un index spécifié. |
| • | Triez les éléments de la collection à l'aide du comparateur par défaut ou d'un comparateur spécifié. |
| • | Réorganisez tout ou partie de la collection. |

L'exemple suivant montre comment utiliser la classe List <T>.

**Utilisation de la classe List <T>**

string s1 = "Latte";

string s2 = "Espresso";

chaîne s3 = "Americano";

string s4 = "Cappuccino";

string s5 = "Moka";

// Ajoute les éléments à une collection fortement typée.

var coffeeBeverages = nouvelle liste <String> ();

coffeeBeverages.Add (s1);

coffeeBeverages.Add (s2);

caféBeverages.Add (s3);

caféBeverages.Add (s4);

caféBeverages.Add (s5);

// Trier les éléments à l'aide du comparateur par défaut.

// Pour les objets de type String, le comparateur par défaut trie les éléments par ordre alphabétique.

caféBeverages.Sort ();

// Ecrit la collection dans une fenêtre de console.

foreach (String coffeeBeverage in coffeeBeverages)

{

Console.WriteLine (coffeeBeverage);

}

Console.ReadLine ("Appuyez sur Entrée pour continuer");

**Autres classes de liste générique**

L'espace de noms System.Collections.Generic comprend également diverses collections génériques qui fournissent des fonctionnalités plus spécialisées:

|  |  |
| --- | --- |
| • | La classe LinkedList <T> fournit une collection générique dans laquelle chaque élément est lié à l'élément précédent de la collection et à l'élément suivant de la collection. Chaque élément de la collection est représenté par un objet LinkedListNode <T>, qui contient une valeur de type T, une référence à l'instance Parent LinkedList <T>, une référence à l'élément précédent de la collection et une référence au suivant élément de la collection. |
| • | La classe Queue <T> représente une collection d'objets fortement typée premier entré, dernier sorti. |
| • | La classe Stack <T> représente une collection d'objets de type dernier entré, dernier sorti fortement typé. |

## **Utilisation des collections de dictionnaires génériques**

Les classes de dictionnaire stockent des collections de paires valeur / clé. La valeur est l'objet que vous souhaitez stocker et la clé est l'objet que vous utilisez pour indexer et récupérer la valeur. Par exemple, vous pouvez utiliser une classe de dictionnaire pour stocker des recettes de café, où la clé est le nom du café et la valeur est la recette de ce café. Dans le cas des dictionnaires génériques, la clé et la valeur sont fortement typées.

**La classe Dictionary <TKey, TValue>**

La classe Dictionary <TKey, TValue> fournit une classe de dictionnaire à usage général et fortement typée. Vous pouvez ajouter des valeurs en double à la collection, mais les clés doivent être uniques. La classe lèvera une ArgumentException si vous essayez d'ajouter une clé qui existe déjà dans le dictionnaire.

L'exemple suivant montre comment utiliser la classe Dictionary <TKey, TValue>:

**Utilisation de la classe Dictionary <TKey, TValue>**

// Crée un nouveau dictionnaire de chaînes avec des clés de chaîne.

var coffeeCodes = nouveau dictionnaire <String, String> ();

// Ajoute quelques entrées au dictionnaire.

coffeeCodes.Add ("CAL", "Café Au Lait");

coffeeCodes.Add ("CSM", "Cinammon Spice Moka");

coffeeCodes.Add ("ER", "Espresso Romano");

coffeeCodes.Add ("RM", "Moka à la framboise");

coffeeCodes.Add ("IC", "Café glacé");

// Cette instruction entraînerait une ArgumentException car la clé existe déjà.

// coffeeCodes.Add ("IC", "Café instantané");

// Pour récupérer la valeur associée à une clé, vous pouvez utiliser l'indexeur.

// Cela lèvera une KeyNotFoundException si la clé n'existe pas.

Console.WriteLine ("La valeur associée à la clé \" CAL \ "est {0}", coffeeCodes ["CAL"]);

// Vous pouvez également utiliser la méthode TryGetValue.

// Ceci renvoie true si la clé existe et false si la clé n'existe pas.

string csmValue = "";

if (coffeeCodes.TryGetValue ("CSM", out csmValue))

{

Console.WriteLine ("La valeur associée à la clé \" CSM \ "est {0}", csmValue);

}

autre

{

Console.WriteLine ("La clé \" CSM \ "n'a pas été trouvée");

}

// Vous pouvez également utiliser l'indexeur pour modifier la valeur associée à une clé.

coffeeCodes ["IC"] = "Café instantané";

**Autres classes de dictionnaires génériques**

Les classes SortedList <TKey, TValue> et SortedDictionary <TKey, TValue> fournissent toutes deux des dictionnaires génériques dans lesquels les entrées sont triées par clé. La différence entre ces classes réside dans l'implémentation sous-jacente:

|  |  |
| --- | --- |
| • | La classe générique SortedList utilise moins de mémoire que la classe générique SortedDictionary. |
| • | La classe SortedDictionary est plus rapide et plus efficace pour insérer et supprimer des données non triées. |

## **Utilisation des interfaces de collecte**

L'espace de noms System.Collections.Generic fournit une gamme de collections génériques adaptées à divers scénarios. Cependant, dans certaines circonstances, vous souhaiterez créer vos propres classes de collection génériques afin de fournir des fonctionnalités plus spécialisées. Par exemple, vous souhaiterez peut-être stocker des données dans une arborescence ou créer une liste liée circulaire.

Par où devez-vous commencer lorsque vous souhaitez créer une classe de collection personnalisée? Toutes les collections ont certaines choses en commun. Par exemple, vous souhaiterez généralement pouvoir énumérer les éléments de la collection à l'aide d'une boucle foreach, et vous aurez besoin de méthodes pour ajouter des éléments, supprimer des éléments et effacer la liste. L'image sur la diapositive montre que le .NET Framework fournit un ensemble hiérarchique d'interfaces qui définissent les caractéristiques et les comportements des collections. Ces interfaces s'appuient les unes sur les autres pour définir progressivement des fonctionnalités plus spécifiques.

**Les interfaces IEnumerable et IEnumerable <T>**

Si vous souhaitez pouvoir utiliser une boucle foreach pour énumérer les éléments de votre collection générique personnalisée, vous devez implémenter l'interface IEnumerable <T>. La méthode IEnumerable <T> définit une méthode unique nommée GetEnumerator (). Cette méthode doit renvoyer un objet de type IEnumerator <T>. L'instruction foreach s'appuie sur cet objet énumérateur pour parcourir la collection.

L'interface IEnumerable <T> hérite de l'interface IEnumerable, qui définit également une méthode unique nommée GetEnumerator (). Lorsqu'une interface hérite d'une autre interface, elle expose tous les membres de l'interface parent. En d'autres termes, si vous implémentez IEnumerable <T>, vous devez également implémenter IEnumerable.

**L'interface ICollection <T>**

L'interface ICollection <T> définit la fonctionnalité de base commune à toutes les collections génériques. L'interface hérite de IEnumerable <T>, ce qui signifie que si vous souhaitez implémenter ICollection <T>, vous devez également implémenter les membres définis par IEnumerable <T> et IEnumerable.

L'interface ICollection <T> définit les méthodes suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Ajouter** | Ajoute un élément de type T à la collection. |
| **Dégager** | Supprime tous les éléments de la collection. |
| **Contient** | Indique si la collection contient une valeur spécifique. |
| **Copier** | Copie les éléments de la collection dans un tableau. |
| **Supprimer** | Supprime un objet spécifique de la collection. |

L'interface ICollection <T> définit les propriétés suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Compter** | Obtient le nombre d'éléments de la collection. |
| **IsReadOnly** | Indique si la collection est en lecture seule. |

**L'interface IList <T>**

L'interface IList <T> définit la fonctionnalité de base pour les classes de liste génériques. Vous devez implémenter cette interface si vous définissez une collection linéaire de valeurs uniques. Outre les membres dont elle hérite de ICollection <T>, l'interface IList <T> définit des méthodes et des propriétés qui vous permettent d'utiliser des indexeurs pour travailler avec les éléments de la collection. Par exemple, si vous créez une liste nommée myList, vous pouvez utiliser myList [0] pour accéder au premier élément de la collection.

L'interface IList <T> définit les méthodes suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Insérer** | Insère un élément dans la collection à l'index spécifié. |
| **RemoveAt** | Supprime l'élément à l'index spécifié de la collection. |

L'interface IList <T> définit les propriétés suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Indice de** | Détermine la position d'un élément spécifié dans la collection. |

**L'interface IDictionary <TKey, TValue>**

L'interface IDictionary <TKey, TValue> définit la fonctionnalité de base pour les classes de dictionnaire génériques. Vous devez implémenter cette interface si vous définissez une collection de paires clé-valeur. Outre les membres dont elle hérite de ICollection <T>, l'interface IDictionary <T> définit des méthodes et des propriétés spécifiques au travail avec des paires clé-valeur.

L'interface IDictionary <TKey, TValue> définit les méthodes suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Ajouter** | Ajoute un élément avec la clé et la valeur spécifiées à la collection. |
| **Contient la clé** | Indique si la collection comprend une paire clé-valeur avec la clé spécifiée. |
| **GetEnumerator** | Renvoie un énumérateur d'objets KeyValuePair <TKey, TValue>. |
| **Supprimer** | Supprime l'élément avec la clé spécifiée de la collection. |
| **TryGetValue** | Tente de définir la valeur d'un paramètre de sortie sur la valeur associée à une clé spécifiée. Si la clé existe, la méthode renvoie true. Si la clé n'existe pas, la méthode renvoie false et le paramètre de sortie reste inchangé. |

L'interface IDictionary <TKey, TValue> définit les propriétés suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Article** | Obtient ou définit la valeur d'un élément de la collection, en fonction d'une clé spécifiée. Cette propriété vous permet d'utiliser la notation d'indexeur, par exemple myDictionary [myKey] = myValue. |
| **Clés** | Renvoie les clés de la collection en tant qu'instance ICollection <T>. |
| **Valeurs** | Renvoie les valeurs de la collection en tant qu'instance ICollection <T>. |

**Liens de référence:**Pour obtenir des informations complètes et des exemples de toutes les interfaces génériques couvertes dans cette rubrique, consultez la page System.Collections.Generic Namespace à l'adresse<https://aka.ms/moc-20483c-m4-pg2>.

## **Création de collections énumérables**

Pour énumérer sur une collection, vous utilisez généralement une boucle foreach. La boucle foreach expose chaque élément de la collection à son tour, dans un ordre approprié à la collection. L'instruction foreach masque certaines des complexités de l'énumération. Pour que l'instruction foreach fonctionne, une classe de collection générique doit implémenter l'interface IEnumerable <T>. Cette interface expose une méthode, GetEnumerator, qui doit renvoyer un type IEnumerator <T>.

**L'interface IEnumerator <T>**

L'interface IEnumerator <T> définit la fonctionnalité que tous les énumérateurs doivent implémenter.

L'interface IEnumerator <T> définit les méthodes suivantes:

| **Nom** | **La description** |
| --- | --- |
| **DéplacerSuivant** | Avance l'énumérateur vers l'élément suivant de la collection. |
| **Réinitialiser** | Définit l'énumérateur sur sa position de départ, qui est avant le premier élément de la collection. |

L'interface IEnumerator <T> définit les propriétés suivantes:

| **Nom** | **La description** |
| --- | --- |
| **Actuel** | Obtient l'élément vers lequel pointe l'énumérateur. |

Un énumérateur est essentiellement un pointeur vers les éléments de la collection. Le point de départ du pointeur est avant le premier élément. Lorsque vous appelez la méthode MoveNext, le pointeur passe à l'élément suivant de la collection. La méthode MoveNext renvoie true si l'énumérateur a pu avancer d'une position, ou false s'il a atteint la fin de la collection. À tout moment au cours de l'énumération, la propriété Current renvoie l'élément vers lequel l'énumérateur pointe actuellement.

Lorsque vous créez un énumérateur, vous devez définir:

|  |  |
| --- | --- |
| • | Quel élément l'énumérateur doit traiter comme le premier élément de la collection. |
| • | Dans quel ordre l'énumérateur doit parcourir les éléments de la collection. |

**L'interface IEnumerable <T>**

L'interface IEnumerable <T> définit une méthode unique nommée GetEnumerator. Cela renvoie une instance IEnumerator <T>.

La méthode GetEnumerator retourne l'énumérateur par défaut pour votre classe de collection. Il s'agit de l'énumérateur qu'une boucle foreach utilisera, sauf si vous spécifiez une alternative. Cependant, vous pouvez créer des méthodes supplémentaires pour exposer d'autres énumérateurs.

L'exemple suivant montre une classe de collection personnalisée qui implémente un énumérateur par défaut, ainsi qu'un autre énumérateur qui énumère la collection dans l'ordre inverse:

**Énumérateurs par défaut et alternatifs**

classe CustomCollection <T>: IEnumerable <T>

{

public IEnumerator <T> Vers l'arrière ()

{

// Cette méthode renvoie un autre énumérateur.

// Les détails d'implémentation ne sont pas affichés.

}

#region IEnumerable <T> Membres

public IEnumerator <T> GetEnumerator ()

{

// Cette méthode renvoie l'énumérateur par défaut.

// Les détails d'implémentation ne sont pas affichés.

}

#endregion

#region IEnumerable Members

IEnumerator IEnumerable.GetEnumerator ()

{

// Cette méthode est requise car IEnumerable <T> hérite de IEnumerable

throw new NotImplementedException ();

}

#endregion

}

L'exemple suivant montre comment utiliser un énumérateur par défaut ou un autre énumérateur pour parcourir une collection:

**Énumération d'une collection**

Numéros de CustomCollection <Int32> = new CustomCollection <Int32> ();

// Ajoute des éléments à la collection.

// Utilisez l'énumérateur par défaut pour parcourir la collection:

foreach (nombre entier en nombres)

{

//…

}

// Utilisez l'énumérateur alternatif pour parcourir la collection:

foreach (nombre entier en nombres.Backwards ())

{

//…

}

**Implémentation de l'énumérateur**

Vous pouvez fournir un énumérateur en créant une classe personnalisée qui implémente l'interface IEnumerator <T>. Toutefois, si votre classe de collection personnalisée utilise un type énumérable sous-jacent pour stocker des données, vous pouvez utiliser un itérateur pour implémenter l'interface IEnumerable <T> sans réellement fournir une implémentation IEnumerator <T>. La meilleure façon de comprendre les itérateurs est de commencer par un exemple simple.

L'exemple suivant montre comment utiliser un itérateur pour implémenter un énumérateur:

**Implémentation d'un énumérateur à l'aide d'un itérateur**

en utilisant le système;

en utilisant System.Collections;

using System.Collections.Generic;

classe BasicCollection <T>: IEnumerable <T>

{

Liste privée <T> data = nouvelle liste <T> ();

public void FillList (paramètres T [] éléments)

{

foreach (var datum dans les éléments)

data.Add (datum);

}

IEnumerator <T> IEnumerable <T> .GetEnumerator ()

{

foreach (var datum dans les données)

{

yield return datum;

}

}

IEnumerator IEnumerable.GetEnumerator ()

{

throw new NotImplementedException ();

}

}

L'exemple montre une classe de collection générique personnalisée qui utilise une instance List <T> pour stocker des données. L'instance List <T> est remplie par la méthode FillList. Lorsque la méthode GetEnumerator est appelée, une boucle foreach énumère la collection sous-jacente. Dans la boucle foreach, une instruction yield return est utilisée pour renvoyer chaque élément de la collection. C'est cette instruction yield return qui définit l'itérateur - essentiellement, l'instruction yield return suspend l'exécution pour renvoyer l'élément actuel à l'appelant avant que l'élément suivant de la séquence ne soit récupéré. De cette façon, bien que la méthode GetEnumerator ne semble pas renvoyer un type IEnumerator, le compilateur est en mesure de créer un énumérateur à partir de la logique d'itération que vous avez fournie.

## **Démonstration: ajout de la validation des données et de la sécurité de type au laboratoire d'application**

Dans cette démonstration, vous découvrirez les tâches que vous effectuerez dans le laboratoire pour ce module.

### **Démonstration étapes**

Vous trouverez les étapes de la démonstration: Ajout de la validation des données et de la sécurité de type à la section Application Lab sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md>.

# **Atelier: Ajout de la validation des données et de la sécurité de type à l'application**

### **Scénario**

Maintenant que les fonctionnalités de navigation de l'interface utilisateur fonctionnent, vous décidez de remplacer les structures simples par des classes pour rendre votre application plus efficace et plus simple.

Il vous a également été demandé d'inclure une logique de validation dans l'application pour vous assurer que lorsqu'un utilisateur ajoute des notes à un étudiant, les données sont valides avant d'être écrites dans la base de données. Vous décidez de créer un projet de test unitaire qui effectuera des tests par rapport à la validation requise pour différents scénarios de notation.

Les enseignants qui ont vu l'application ont exprimé leur inquiétude que les élèves de leurs classes soient affichés dans un ordre aléatoire. Vous décidez d'utiliser l'interface IComparable pour leur permettre d'être affichés par ordre alphabétique.

Enfin, il vous a été demandé d'ajouter des fonctionnalités à l'application pour permettre aux enseignants d'ajouter et de supprimer des élèves d'une classe, et d'ajouter les notes des élèves à la base de données.

### **Objectifs**

Après avoir terminé cet atelier, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Créez des classes. |
| • | Rédiger le code de validation des données et les tests unitaires. |
| • | Implémentez l'interface IComparable. |
| • | Utilisez des collections génériques. |

##### **Configuration du laboratoire**

Durée estimée: 75 minutes

Vous trouverez les étapes de haut niveau sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAB_MANUAL.md>.

Vous trouverez les étapes détaillées sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAK.md>.

### **Exercice 1: Implémentation des structures enseignant, élève et scolaire en tant que classes**

##### **Scénario**

Dans cet exercice, vous allez convertir les structures existantes Teacher, Student et Grade en classes. Cela vous permettra d'implémenter les fonctionnalités supplémentaires requises pour chaque classe, telles que l'ajout de constructeurs, de propriétés et de méthodes. Dans les classes Enseignant et Etudiant, vous allez rendre la propriété de mot de passe en écriture seule, ajouter la méthode VerifyPassword, puis définir leurs constructeurs respectifs. Vous allez également modifier la méthode Logon\_Click pour utiliser la méthode VerifyPassword pour vérifier les mots de passe lorsqu'un utilisateur ouvre une session. Enfin, vous exécuterez l'application et vérifierez qu'elle fonctionne toujours correctement, permettant à un élève ou à un enseignant de se connecter.

**Résultat**: Une fois cet exercice terminé, les structures Teacher, Student et Grade seront implémentées en tant que classes et la méthode VerifyPassword sera appelée lorsqu'un utilisateur se connectera.

### **Exercice 2: Ajout de la validation des données à la classe de qualité**

##### **Scénario**

Dans cet exercice, vous allez définir une liste publique de chaînes appelées Sujets pour contenir les noms de chaque sujet pouvant être évalué, puis la remplir avec des noms de sujet valides. Vous ajouterez ensuite une logique de validation à la classe Note pour vous assurer que le nom du sujet apparaît dans la liste que vous avez créée et que la date et la note d'évaluation contiennent des valeurs autorisées. Enfin, vous allez créer un projet de test unitaire pour vérifier que votre code de validation fonctionne comme prévu.

**Résultat**: Après avoir terminé cet exercice, la classe Grade contiendra une logique de validation.

### **Exercice 3: Affichage des étudiants par ordre de nom**

##### **Scénario**

Dans cet exercice, vous allez écrire du code pour afficher les élèves dans l'ordre alphabétique du nom de famille, puis du prénom.

L'application affiche actuellement les élèves sans ordre spécifique lorsqu'ils sont connectés en tant qu'enseignant, mais vous voulez maintenant qu'ils soient affichés dans l'ordre alphabétique du nom et du prénom. Pour ce faire, vous décidez que la classe Student doit implémenter l'interface IComparable <> pour permettre la comparaison des données des étudiants. Vous pouvez ensuite ajouter du code à la méthode CompareTo dans la classe Student, ce qui permet aux étudiants d'être triés en fonction de leur nom et prénom. Actuellement, les étudiants sont stockés dans une collection ArrayList non sécurisée par type. Vous décidez de modifier cela afin qu'ils soient stockés dans une collection List de type sécurisé. Enfin, vous trierez les données des étudiants, puis exécuterez l'application pour vérifier que les étudiants sont récupérés et affichés dans l'ordre alphabétique de leur nom et prénom.

**Résultat**: Une fois cet exercice terminé, l'application affichera les élèves dans l'ordre alphabétique du nom de famille, puis du prénom.

### **Exercice 4: Permettre aux enseignants de modifier les données de classe et de note**

##### **Scénario**

Dans cet exercice, vous allez écrire un code qui permet à un enseignant d'ajouter un élève, puis de l'inscrire dans une classe. Cela sera mis en œuvre en deux étapes distinctes, car un enseignant peut souhaiter ajouter un élève avant de savoir à quelle classe il sera inscrit. Vous autoriserez également un enseignant à supprimer un élève d'une classe. Lors de l'ajout ou de la suppression d'un élève, vous afficherez une invite pour confirmer que l'enseignant souhaite exécuter l'action.

Pour inscrire un élève dans une classe ou le supprimer d'une classe, vous modifiez la propriété TeacherID de cet élève. L'application comprend désormais la fenêtre AssignStudentDialog, qui affiche une liste des étudiants qui ne sont pas affectés à une classe. Vous devez ajouter du code dans cette fenêtre pour affecter un élève à la classe de l'enseignant et mettre à jour la liste des élèves le cas échéant. Vous devez également ajouter du code pour supprimer un élève d'une classe et pour permettre aux enseignants d'ajouter des notes à leurs élèves. Une fois qu'un étudiant a été ajouté à la base de données, cet étudiant pourra se connecter pour afficher ses propres notes.

**Résultat**: Une fois cet exercice terminé, l'application permettra aux enseignants d'ajouter et de supprimer des élèves de leurs classes et d'ajouter des notes aux élèves.

# **Revue du module et points à retenir**

Dans ce module, vous avez appris à travailler avec des classes, des interfaces et des collections génériques dans Visual C #.

### **Question (s) de révision**