Module 5: Création d'une hiérarchie de classes à l'aide de l'héritage

# **Contenu:**

|  |  |
| --- | --- |
|  | [Présentation du module](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C05.html#P1) |
| **Leçon 1:** | [Création de hiérarchies de classes](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C05.html#P2) |
| **Leçon 2:** | [Extension des classes .NET Framework](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C05.html#P3) |
| **Laboratoire:** | [Refactorisation des fonctionnalités communes dans la classe d'utilisateurs](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C05.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/20483C05.html#P5) |

# **Présentation du module**

Le concept d'héritage est au cœur de la programmation orientée objet dans n'importe quel langage. C'est également l'un des outils les plus puissants de votre boîte à outils de développeur. Essentiellement, l'héritage vous permet de créer de nouvelles classes en héritant des caractéristiques et des comportements des classes existantes. Lorsque vous héritez d'une classe existante et ajoutez vos propres fonctionnalités, votre classe devient une instance plus spécialisée de la classe existante. Non seulement cela vous fait gagner du temps en réduisant la quantité de code à écrire, mais cela vous permet également de créer des hiérarchies de classes associées que vous pouvez ensuite utiliser de manière interchangeable, en fonction de vos besoins.

Dans ce module, vous apprendrez à utiliser l'héritage pour créer des hiérarchies de classes et étendre les types .NET Framework.

### **Objectifs**

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

|  |  |
| --- | --- |
| • | Créez des classes de base et des classes dérivées à l'aide de l'héritage. |
| • | Créez des classes qui héritent des classes .NET Framework. |

# Leçon 1: Création de hiérarchies de classes

Plutôt que de créer de nouvelles classes à partir de rien, dans de nombreux cas, vous pouvez utiliser une classe existante comme base pour votre nouvelle classe. C'est ce qu'on appelle l'héritage. Votre classe hérite de tous les membres de la classe de base et vous incluez simplement la fonctionnalité que vous souhaitez ajouter aux capacités de la classe de base. De cette façon, votre classe devient une version plus spécialisée de la classe de base. Ce concept d'héritage est l'un des principaux piliers de la programmation orientée objet.

Dans cette leçon, vous apprendrez à utiliser l'héritage pour créer des hiérarchies de classes.

## **Objectifs de la leçon**

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

|  |  |
| --- | --- |
| • | Décrivez l'héritage. |
| • | Créez des classes de base. |
| • | Créez des membres de classe de base. |
| • | Créez des classes qui héritent des classes de base. |
| • | Appelez des méthodes de classe de base et des constructeurs à partir d'une classe dérivée. |

## **Qu'est-ce que l'héritage?**

Dans Visual C #, une classe peut hériter d'une autre classe. Lorsque vous créez une classe qui hérite d'une autre classe, votre classe est appelée classe dérivée et la classe dont vous héritez est appelée classe de base. La classe dérivée hérite de tous les membres de la classe de base, y compris les constructeurs, les méthodes, les propriétés, les champs et les événements.

L'héritage vous permet de créer des hiérarchies de classes de plus en plus spécialisées. Plutôt que de créer une classe à partir de rien, vous pouvez hériter d'une classe de base plus générale pour fournir un point de départ pour vos fonctionnalités. L'héritage peut aider à simplifier la maintenance de votre code.

Par exemple, vous définissez une classe nommée Beverage, comme indiqué ci-dessous:

**La classe de boisson**

Boisson de classe publique

{

protégé int servantTemperature;

chaîne publique Name {get; ensemble; }

public bool IsFairTrade {obtenir; ensemble; }

public int GetServingTemperature ()

{

retour serviceTemperature;

}

}

Vous souhaitez maintenant créer une classe pour représenter les cafés. Le café est un type de boisson. Il partage toutes les caractéristiques et comportements d'une boisson. Plutôt que de créer une classe à partir de zéro pour représenter les cafés, vous pouvez créer une classe qui hérite de la classe Beverage. La classe dérivée hérite de tous les membres de la classe Beverage, tels que le champ serveTemperature, la propriété Name, la propriété IsFairTrade et la méthode GetServingTemperature. Dans la classe dérivée, il vous suffit d'ajouter des membres spécifiques aux cafés.

L'exemple suivant montre comment créer une classe pour les cafés qui hérite de la classe Beverage:

**La classe de café**

Café en classe publique: Boisson

{

chaîne publique Bean {get; ensemble; }

chaîne publique Roast {get; ensemble; }

chaîne publique CountryOfOrigin {get; ensemble; }

}

**Noter:**Dans la programmation orientée objet, les termes dérive et hérite sont utilisés de manière interchangeable. Dire que la classe Coffee dérive de la classe Beverage signifie la même chose que dire que la classe Coffee hérite de la classe Beverage.

Comme vous pouvez le voir dans les exemples précédents, la syntaxe d'héritage d'une classe est similaire à la syntaxe d'implémentation d'une interface. En effet, l'héritage d'une classe et l'implémentation d'une interface sont deux exemples d'héritage. Toutefois, vous n'avez pas besoin de dupliquer les membres de la classe de base dans la classe dérivée. En ajoutant la classe de base à votre déclaration de classe, vous mettez tous les membres de la classe de base à la disposition des consommateurs de votre classe dérivée.

L'exemple suivant montre comment utiliser des membres de classe de base dans une classe dérivée:

**Appel des membres de la classe de base**

Café café1 = nouveau café ();

// Utilise les membres de la classe de base.

coffee1.Name = "Quatrième espresso";

coffee1.IsFairTrade = vrai;

int portionTemp = coffee1.GetServingTemperature ();

// Utilise des membres de classe dérivés.

coffee1.Bean = "Arabica";

coffee1.Roast = "Sombre";

coffee1.CountryOfOrigin = "Colombie";

Comme indiqué dans l'exemple ci-dessus, vous pouvez appeler les membres d'une classe de base de la même manière que vous appelez les membres de la classe elle-même.

## **Créer des classes de base**

Lorsque vous créez une classe, vous devez déterminer si vous ou d'autres développeurs devrez l'utiliser comme base pour les classes dérivées. Vous contrôlez totalement si et comment votre classe peut être héritée.

**Classes abstraites et membres**

Dans le cadre d'une conception orientée objet, vous souhaiterez peut-être créer des classes qui servent uniquement de classes de base pour d'autres types. La classe de base peut contenir des fonctionnalités manquantes ou incomplètes et vous ne souhaiterez peut-être pas activer le code pour instancier directement votre classe. Dans ces scénarios, vous pouvez ajouter le mot-clé abstract à votre déclaration de classe.

L'exemple suivant montre comment déclarer une classe abstraite:

**Déclarer une classe abstraite**

boisson de classe abstraite

{

}

Le mot-clé abstract indique que la classe ne peut pas être instanciée directement; il existe uniquement pour définir ou implémenter des membres pour des classes dérivées. Si vous essayez d'instancier une classe abstraite, vous obtiendrez une erreur lorsque vous construirez votre code.

Une classe abstraite peut contenir à la fois des membres abstraits et non abstraits. Les membres abstraits sont également déclarés avec le mot-clé abstract et sont conceptuellement similaires aux membres d'interface, en ce sens que les membres abstraits définissent la signature du membre mais ne fournissent aucun détail d'implémentation. Toute classe qui hérite de la classe abstraite doit fournir une implémentation pour les membres abstraits. Les membres non abstraits, cependant, peuvent être utilisés directement par des classes dérivées.

L'exemple suivant illustre la différence entre les membres abstraits et non abstraits:

**Membres abstraits et non-abstraits**

boisson de classe abstraite

{

// Membres non abstraits.

// Les classes dérivées peuvent utiliser ces membres sans les modifier.

chaîne publique Name {get; ensemble; }

public bool IsFairTrade {obtenir; ensemble; }

// Membres abstraits.

// Les classes dérivées doivent remplacer et implémenter ces membres.

résumé public int GetServingTemperature ();

}

**Noter:**Vous ne pouvez inclure des membres abstraits que dans des classes abstraites. Une classe non abstraite ne peut pas inclure de membres abstraits.

**Classes scellées**

Vous souhaiterez peut-être créer des classes dont vous ne pouvez pas hériter. Vous pouvez empêcher les développeurs d'hériter de vos classes en marquant la classe avec le mot-clé scellé.

L'exemple suivant montre comment utiliser le modificateur scellé:

**Créer une classe scellée**

Classe publique scellée Thé: Boisson

{

//…

}

Vous pouvez appliquer le modificateur scellé aux classes qui héritent d'autres classes et classes qui implémentent des interfaces. Vous ne pouvez pas utiliser le modificateur scellé et le modificateur abstrait sur la même classe, car une classe scellée est l'opposé d'une classe abstraite - une classe scellée ne peut pas être héritée, alors qu'une classe abstraite doit être héritée.

Toute classe statique est également une classe scellée. Vous ne pouvez jamais hériter d'une classe statique. De même, les membres statiques des classes non statiques ne sont pas hérités par les classes dérivées.

## **Création de membres de classe de base**

Vous souhaiterez peut-être implémenter une méthode dans votre classe de base, mais autorisez les classes dérivées à remplacer votre implémentation de méthode par des fonctionnalités plus spécifiques. Pour créer un membre que les développeurs peuvent remplacer dans une classe dérivée, vous utilisez le mot clé virtual.

L'exemple suivant montre comment créer une méthode virtuelle dans une classe:

**Ajout de membres virtuels**

Boisson de classe publique

{

protégé int servantTemperature;

chaîne publique Name {get; ensemble; }

public bool IsFairTrade {obtenir; ensemble; }

public virtual int GetServingTemperature ()

{

// Il s'agit de l'implémentation par défaut de la méthode GetServingTemperature.

// Étant donné que la méthode est déclarée virtuelle, vous pouvez remplacer l'implémentation dans les classes dérivées.

retour serviceTemperature;

}

}

Lorsque vous créez une classe, vous pouvez utiliser des modificateurs d'accès pour contrôler si les membres de votre classe sont accessibles aux types dérivés. Vous pouvez utiliser les modificateurs d'accès suivants pour les membres de la classe:

| **Modificateur d'accès** | **Des détails** |
| --- | --- |
| **Publique** | Le membre est disponible pour le code exécuté dans n'importe quel assembly. |
| **protégé** | Le membre est disponible uniquement dans la classe conteneur ou dans les classes dérivées de la classe conteneur. |
| **interne** | Le membre n'est disponible que pour coder dans l'assembly actuel. |
| **intérieur protégé** | Le membre est disponible pour n'importe quel code dans l'assembly actuel et pour les types dérivés de la classe conteneur dans n'importe quel assembly. |
| **privé** | Le membre est disponible uniquement dans la classe conteneur. |
| **privé protégé** | Le membre est disponible pour les types dérivés de la classe conteneur, mais uniquement dans son assembly conteneur. (Uniquement disponible depuis Visual C # 7.2) |

Les membres d'une classe sont privés par défaut. Les membres privés ne sont pas hérités par les classes dérivées. Si vous voulez que les membres qui seraient autrement privés soient accessibles aux classes dérivées, vous devez préfixer le membre avec le mot-clé protected.

## **Hériter d'une classe de base**

Pour hériter d'une autre classe, vous ajoutez un signe deux-points suivi du nom de la classe de base à votre déclaration de classe.

L'exemple suivant montre comment hériter d'une classe de base:

**Déclaration d'une classe qui hérite d'une classe de base**

Café en classe publique: Boisson

{

}

La classe dérivée hérite de chaque membre de la classe de base. Dans votre classe dérivée, vous pouvez ajouter de nouveaux membres pour étendre les fonctionnalités du type de base.

Une classe ne peut hériter que d'une classe de base. Cependant, votre classe peut implémenter une ou plusieurs interfaces en plus de dériver d'un type de base.

**Remplacement des membres de la classe de base**

Dans certains cas, vous souhaiterez peut-être modifier la façon dont un membre de classe de base fonctionne dans votre classe dérivée.

Par exemple, la classe de base Beverage comprend une méthode nommée GetServingTemperature:

**Ajout de membres virtuels**

Boisson de classe publique

{

protégé int servantTemperature;

public virtual int GetServingTemperature ()

{

retour serviceTemperature;

}

// Les autres membres de la classe ne sont pas affichés.

}

Étant donné que GetServingTemperature est une méthode virtuelle, vous pouvez la remplacer dans une classe dérivée. Pour remplacer une méthode virtuelle dans une classe dérivée, vous créez une méthode avec la même signature et lui préfixez le mot-clé override.

L'exemple suivant montre comment remplacer une méthode virtuelle dans une classe dérivée:

**Remplacement d'une méthode virtuelle à l'aide du mot-clé override**

Café en classe publique: Boisson

{

protected bool includesMilk;

private int servantTempWithMilk;

private int servantTempWithoutMilk;

substitution publique int GetServingTemperature ()

{

if (includesMilk) retourne serveTempWithMilk

else retourne serveTempWithoutMilk;

}

}

Vous pouvez utiliser la même approche pour remplacer les propriétés, les indexeurs et les événements. Dans chaque cas, vous ne pouvez remplacer un membre de classe de base que si le membre est marqué comme virtuel dans la classe de base. Vous ne pouvez pas remplacer les constructeurs.

Vous pouvez également remplacer un membre de classe de base en utilisant le nouveau mot clé:

**Remplacement d'une méthode virtuelle à l'aide du nouveau mot-clé**

Café en classe publique: Boisson

{

public new int GetServingTemperature ()

{

//…

}

}

Lorsque vous utilisez le mot clé override, votre méthode étend la méthode de classe de base. En revanche, lorsque vous utilisez le nouveau mot-clé, votre méthode masque la méthode de la classe de base. Cela entraîne des différences subtiles mais importantes dans la façon dont le compilateur traite votre classe de base et les types de classe dérivés.

**Remarque: remplacer**peut être utilisé pour les membres virtuels et abstraits. La différence est que le remplacement des membres virtuels est facultatif, tandis que le remplacement des membres abstraits est obligatoire. Seule une classe abstraite doit implémenter chaque membre abstrait défini par ses ancêtres.

**Liens de référence:**Pour une explication détaillée des différences de comportement entre le mot-clé override et le nouveau mot-clé, consultez Savoir quand utiliser le remplacement et les nouveaux mots-clés (Guide de programmation C #) à l'adresse<https://aka.ms/moc-20483c-m5-pg1>.

**Sceller les membres remplacés**

Lorsque vous substituez un membre de classe de base, vous pouvez empêcher les classes qui dérivent de votre classe de remplacer votre implémentation du membre de classe de base à l'aide du mot clé scellé.

L'exemple suivant montre comment sceller un membre de classe de base substitué:

**Sceller un membre remplacé**

Café en classe publique: Boisson

{

remplacement public scellé int GetServingTemperature ()

{

// Les classes dérivées ne peuvent pas remplacer cette méthode.

}

}

En scellant un membre surchargé, vous forcez toutes les classes qui dérivent de votre classe à utiliser votre implémentation du membre de classe de base, plutôt que de créer la leur. Cela peut être utile lorsque vous devez contrôler le comportement de vos classes et vous assurer que les classes dérivées n'essaient pas de modifier le fonctionnement de membres spécifiques.

Vous ne pouvez appliquer le modificateur scellé à un membre que si le membre est un membre de remplacement. N'oubliez pas que les membres sont intrinsèquement scellés à moins qu'ils ne soient marqués comme virtuels. Dans ce cas, étant donné que la méthode de classe de base est marquée comme virtuelle, tous les descendants peuvent remplacer la méthode à moins que vous ne la scelliez à un moment donné dans la hiérarchie des classes.

## **Appel de constructeurs et de membres de classe de base**

Vous pouvez utiliser le mot-clé base pour accéder aux méthodes de classe de base et aux constructeurs à partir d'une classe dérivée. Ceci est utile dans les scénarios suivants:

|  |  |
| --- | --- |
| • | Vous remplacez une méthode de classe de base, mais vous souhaitez toujours exécuter la fonctionnalité dans la méthode de classe de base en plus de vos propres fonctionnalités supplémentaires. |
| • | Vous créez une nouvelle méthode, mais vous souhaitez appeler une méthode de classe de base dans le cadre de votre logique de méthode. |
| • | Vous créez un constructeur et vous souhaitez appeler un constructeur de classe de base dans le cadre de votre logique d'initialisation. |
| • | Vous souhaitez appeler une méthode de classe de base à partir d'un accesseur de propriété. |

**Appel de constructeurs de classe de base à partir d'une classe dérivée**

Vous ne pouvez pas remplacer les constructeurs dans les classes dérivées. Au lieu de cela, lorsque vous créez des constructeurs dans une classe dérivée, vos constructeurs appelleront automatiquement le constructeur de classe de base par défaut avant d'exécuter l'une des logiques que vous avez ajoutées. Cependant, dans certaines circonstances, vous souhaiterez peut-être que vos constructeurs appellent un autre constructeur de classe de base. Dans ces cas, vous pouvez utiliser le mot clé base dans vos déclarations de constructeur pour spécifier le constructeur de classe de base que vous souhaitez appeler.

L'exemple suivant montre comment appeler des constructeurs de classe de base:

**Appel des constructeurs de classe de base**

Boisson de classe publique

{

Boisson publique ()

{

// Détails d'implémentation non affichés.

}

public Beverage (nom de chaîne, booléen isFairTrade, int portionTemp)

{

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

}

// Les autres membres de la classe ne sont pas affichés.

}

Café en classe publique: Boisson

{

Café public ()

{

// Ce constructeur appelle implicitement le constructeur Beverage par défaut.

// La déclaration est implicitement équivalente à public Coffee (): base ()

// Vous pouvez inclure du code supplémentaire ici.

}

café public (nom de chaîne, bool isFairTrade, int serveTemp, string bean, string roast)

: base (nom, isFairTrade, servantTemp)

{

// Ceci appelle le constructeur Beverage (string, bool, int).

// Vous pouvez inclure du code supplémentaire ici:

Haricot = haricot;

Rôti = rôti;

}

chaîne publique Bean {get; ensemble; }

chaîne publique Roast {get; ensemble; }

chaîne publique CountryOfOrigin {get; ensemble; }

}

Comme le montre l'exemple précédent, vous devez appeler le constructeur de classe de base à partir de votre déclaration de constructeur. Vous ne pouvez pas appeler le constructeur de classe de base à partir du corps de votre méthode constructeur. Vous pouvez utiliser les paramètres nommés de votre déclaration de constructeur comme arguments du constructeur de classe de base.

**Appel de méthodes de classe de base à partir d'une classe dérivée**

Vous pouvez appeler des méthodes de classe de base à partir de corps de méthode ou d'accesseurs de propriété dans une classe dérivée. Pour ce faire, vous utilisez le mot-clé de base comme vous utiliseriez une variable d'instance.

L'exemple suivant montre comment appeler des méthodes de classe de base:

**Appel de méthodes de classe de base**

Boisson de classe publique

{

protégé int servantTemperature;

public virtual int GetServingTemperature ()

{

retour serviceTemperature;

}

// Les constructeurs et les membres de classe supplémentaires ne sont pas affichés.

}

Café en classe publique: Boisson

{

bool glacé = faux;

protected int portionTempIced = 40;

substitution publique int GetServingTemperature ()

{

si (glacé)

{

return serveTempIced;

}

autre

{

return base.GetServingTemperature ();

}

}

}

N'oubliez pas que les règles d'héritage ne s'appliquent pas aux classes et aux membres statiques. En tant que tel, vous ne pouvez pas utiliser le mot clé de base dans une méthode statique.

## **Démonstration: appeler des constructeurs de classe de base**

Dans cette démonstration, vous parcourrez l'exécution d'une application. La solution inclut une classe nommée Coffee qui hérite d'une classe nommée Beverage. La classe Coffee comprend deux constructeurs: un qui appelle implicitement le constructeur de classe de base par défaut et un qui appelle explicitement un constructeur de classe de base alternatif. L'application crée des instances de la classe Coffee à l'aide de ces deux constructeurs. Dans les deux cas, vous pouvez observer comment les constructeurs de classe dérivés appellent les constructeurs de classe de base. Vous verrez également comment les constructeurs de classe dérivés transmettent les valeurs d'argument aux constructeurs de classe de base.

### **Démonstration étapes**

Vous trouverez les étapes dans la section Démonstration: Appel de constructeurs de classes de base sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md>.

# Leçon 2: Extension des classes .NET Framework

Le .NET Framework contient plusieurs milliers de classes qui fournissent un large éventail de fonctionnalités. Lorsque vous créez vos propres classes, vous devez chercher à construire sur ces classes en héritant des types .NET Framework dans la mesure du possible. Non seulement cela réduit la quantité de code à écrire, mais cela permet également de garantir que vos cours fonctionnent de manière standardisée.

Le .NET Framework vous permet également de créer des méthodes d'extension pour ajouter des fonctionnalités aux types .NET Framework scellés. Cela vous permet d'étendre les fonctionnalités des types intégrés, tels que la classe String, lorsque l'approche d'héritage n'est pas autorisée.

Dans cette leçon, vous apprendrez à étendre les types .NET Framework à l'aide des méthodes d'héritage et d'extension.

## **Objectifs de la leçon**

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

|  |  |
| --- | --- |
| • | Créez des classes qui héritent des types .NET Framework. |
| • | Créez des classes d'exceptions personnalisées. |
| • | Lancez et interceptez des exceptions personnalisées. |
| • | Créez des classes qui héritent des types génériques. |
| • | Créez des méthodes d'extension pour les types .NET Framework. |

## **Hériter des classes .NET Framework**

Il existe près de 15 000 types publics dans le .NET Framework. Bien que toutes ces classes ne soient pas extensibles, beaucoup d’entre elles le sont. Lorsque vous souhaitez développer une classe, dans de nombreux cas, il existe une classe .NET Framework intégrée qui peut fournir une base pour votre code.

Il y a deux avantages clés à créer une classe qui hérite d'une classe .NET Framework, plutôt que de développer une classe à partir de zéro:

|  |  |
| --- | --- |
| • | **Temps de développement réduit**. En héritant d'une classe existante, vous réduisez la quantité de logique que vous devez créer vous-même. |
| • | **Fonctionnalité standardisée**. Tout comme l'implémentation d'une interface, hériter d'une classe de base standard signifie que votre classe fonctionnera de manière standardisée. Vous pouvez également représenter des instances de votre classe en tant qu'instances de la classe de base, ce qui permet aux développeurs d'utiliser plus facilement votre classe avec d'autres types qui dérivent de la même classe de base. |

Les règles d'héritage s'appliquent aux classes .NET Framework intégrées de la même manière qu'elles s'appliquent aux classes personnalisées:

|  |  |
| --- | --- |
| • | Vous pouvez créer une classe qui dérive d'une classe .NET Framework, à condition que la classe ne soit pas scellée ou statique. |
| • | Vous pouvez remplacer tous les membres de classe de base marqués comme virtuels. |
| • | Si vous héritez d'une classe abstraite, vous devez fournir des implémentations pour tous les membres abstraits. |

Lorsque vous créez une classe, sélectionnez une classe de base qui minimise la quantité de codage et de personnalisation requise. Si vous vous trouvez en train de répliquer des fonctionnalités disponibles dans les classes intégrées, vous devriez probablement choisir une classe de base plus spécifique. D'un autre côté, si vous trouvez que vous devez remplacer plusieurs membres, vous devriez probablement choisir une classe de base plus générale.

Par exemple, considérez que vous souhaitez créer une classe qui stocke une liste linéaire de valeurs. La classe doit vous permettre de supprimer les éléments en double de la liste. Plutôt que de créer une nouvelle classe de liste à partir de rien, vous pouvez accomplir cela en créant une classe qui hérite de la classe générique List <T> et en ajoutant une méthode unique pour supprimer les éléments en double. De plus, vous pouvez tirer parti de la méthode Sort dans la classe List <T>. Si vous appelez la méthode Sort, tous les éléments en double seront adjacents dans la collection, ce qui peut faciliter leur identification et leur suppression.

L'exemple suivant montre comment étendre la classe List <T>:

**Extension d'une classe .NET Framework**

classe publique UniqueList <T>: List <T>

{

public void RemoveDuplicates ()

{

base.Sort ();

pour (int i = this.Count - 1; i> 0; i--)

{

if (this [i] .Equals (this [i-1]))

{

this.RemoveAt (i);

}

}

}

}

Lorsque vous utilisez cette approche, les consommateurs de votre classe ont accès à toutes les fonctionnalités fournies par la classe de base List <T>. Ils ont également accès à la méthode RemoveDuplicates supplémentaire que vous avez fournie dans votre classe dérivée.

## **Création d'exceptions personnalisées**

Le .NET Framework contient des classes d'exception intégrées pour représenter les conditions d'erreur les plus courantes. Par exemple:

|  |  |
| --- | --- |
| • | Si vous appelez une méthode avec une valeur d'argument null et que la méthode ne peut pas gérer les valeurs d'argument nul, la méthode lèvera une ArgumentNullException. |
| • | Si vous essayez de diviser une valeur numérique par zéro, le runtime lèvera une DivideByZeroException. |
| • | Si vous essayez de récupérer un élément indexé d'une collection et l'indexer en dehors des limites de la collection, l'indexeur lèvera une IndexOutOfRangeException. |

**Noter:**La plupart des classes d'exceptions intégrées sont définies dans l'espace de noms System. Pour plus d'informations sur l'espace de noms système, consultez la page Espace de noms système à l'adresse<https://aka.ms/moc-20483c-m5-pg2>.

Lorsque vous devez lever des exceptions dans votre code, vous devez réutiliser les types d'exceptions .NET Framework existants dans la mesure du possible. Toutefois, dans certaines circonstances, vous souhaiterez peut-être créer vos propres types d'exceptions personnalisés.

**Quand devez-vous créer un type d'exception personnalisé?**

Vous devez envisager de créer un type d'exception personnalisé lorsque:

|  |  |
| --- | --- |
| • | Les types d'exceptions existants ne représentent pas correctement la condition d'erreur que vous identifiez. |
| • | L'exception nécessite une action corrective très spécifique qui diffère de la façon dont vous géreriez les types d'exception intégrés. |

N'oubliez pas que l'objectif principal des types d'exception est de vous permettre de gérer des conditions d'erreur spécifiques de manière spécifique en interceptant un type d'exception spécifique. Le type d'exception n'est pas conçu pour communiquer les détails précis du problème. Toutes les classes d'exception incluent une propriété de message à cet effet. Par conséquent, vous ne devez pas créer une classe d'exception personnalisée uniquement pour communiquer la nature d'une condition d'erreur. Créez une classe d'exception personnalisée uniquement si vous devez gérer cette condition d'erreur de manière distincte.

**Création de types d'exception personnalisés**

Toutes les classes d'exception dérivent finalement de la classe System.Exception. Cette classe fournit une gamme de propriétés que vous pouvez utiliser pour fournir plus de détails sur la condition d'erreur. Par exemple:

|  |  |
| --- | --- |
| • | La propriété Message vous permet de fournir plus d'informations sur ce qui s'est passé sous forme de chaîne de texte. |
| • | La propriété InnerException vous permet d'identifier une autre instance d'exception à l'origine de l'instance actuelle. |
| • | La propriété Source vous permet de spécifier l'élément ou l'application qui a provoqué la condition d'erreur. |
| • | La propriété Data vous permet de fournir plus d'informations sur la condition d'erreur sous forme de paires clé-valeur. |

Lorsque vous créez un type d'exception personnalisé, vous devez utiliser ces propriétés existantes dans la mesure du possible, plutôt que de créer vos propres propriétés alternatives. À un niveau élevé, le processus de création d'une classe d'exceptions personnalisée est le suivant:

|  |  |
| --- | --- |
| 1. | Créez une classe qui hérite de la classe System.Exception. |
| 2. | Mappez vos constructeurs de classe aux constructeurs de classe de base. |
| 3. | Ajoutez des membres si nécessaire. |

L'exemple suivant montre comment créer une classe d'exception personnalisée:

**Création d'un type d'exception personnalisé**

en utilisant le système;

public class LoyaltyCardNotFoundException: exception

{

public LoyaltyCardNotFoundException ()

{

// Ceci appelle implicitement le constructeur de classe de base.

}

public LoyaltyCardNotFoundException (message de chaîne): base (message)

{

}

public LoyaltyCardNotFoundException (message de chaîne, exception interne): base (message, interne)

{

}

}

**Noter:**Lorsque vous créez une classe d'exception personnalisée, il est recommandé d'inclure le mot Exception à la fin du nom de votre classe.

## **Lancer et intercepter des exceptions personnalisées**

Une fois que vous avez créé votre type d'exception personnalisé, vous pouvez lever et intercepter des exceptions personnalisées de la même manière que vous le feriez pour toutes les autres exceptions. Pour lever une exception personnalisée, vous utilisez le mot clé throw et créez une nouvelle instance de votre type d'exception.

Le code suivant montre comment vous pouvez lever une exception personnalisée:

**Lancer une exception personnalisée**

publique LoyaltyCard

{

public static int GetBalance (chaîne de fidélitéCardNumber)

{

var client = LoyaltyCard.GetCustomer (fidélitéCardNumber);

if (client == null)

{

throw new LoyaltyCardNotFoundException ("Le numéro de carte fourni n'a pas été trouvé");

}

autre

{

retour client.TotalPoints;

}

}

// Les autres membres de la classe ne sont pas affichés.

}

Pour intercepter l'exception, vous utilisez un bloc try / catch. N'oubliez pas que vous devez toujours essayer d'attraper les exceptions les plus spécifiques en premier, et l'exception la plus générale (généralement System.Exception) en dernier.

L'exemple suivant montre comment vous pouvez intercepter une exception personnalisée:

**Attraper une exception personnalisée**

public bool PayWithPoints (int costInPoints, string cardNumber)

essayer

{

int totalPoints = LoyaltyCard.GetBalance (cardNumber);

// Lance une exception LoyaltyCardNotFoundException si le numéro de carte n'est pas valide.

if (totalPoints> = costInPoints)

{

LoyaltyCard.DeductPoints (costInPoints);

retourne vrai;

}

sinon retourner faux;

}

catch (LoyaltyCardNotFoundException)

{

// Prenez les mesures appropriées pour remédier au numéro de carte invalide.

retourne faux;

}

catch (Exception)

{

// Attrape d'autres exceptions imprévues.

retourne faux;

}

## **Hériter de types génériques**

Lorsque vous héritez d'une classe générique, vous devez décider comment vous souhaitez gérer les paramètres de type de la classe de base. Vous pouvez gérer les paramètres de type de deux manières:

|  |  |
| --- | --- |
| • | Laissez le paramètre de type du type de base non spécifié. |
| • | Spécifiez un argument de type pour le type de base. |

Prenons un exemple où vous souhaitez créer une classe de liste personnalisée qui hérite de List <T>. Si vous laissez le paramètre de type du type de base non spécifié, vous devez inclure le même paramètre de type dans votre déclaration de classe.

L'exemple suivant montre comment hériter d'un type de base générique sans spécifier d'argument de type:

**Hériter d'un type de base générique sans spécifier d'argument de type**

classe publique CustomList <T>: List <T>

{}

Dans l'exemple ci-dessus, lorsque vous instanciez la classe CustomList et fournissez un argument de type pour T, le même argument de type est appliqué à la classe de base.

Vous pouvez également spécifier un argument de type pour le type de base dans votre déclaration de classe. Lorsque vous utilisez cette approche, toutes les références au paramètre de type dans le type de base sont remplacées par le type que vous spécifiez dans votre déclaration de classe.

L'exemple suivant montre comment spécifier un argument de type pour un type de base:

**Hériter d'un type de base générique en spécifiant un argument de type**

Public class CustomList: List <int>

{}

Dans l'exemple ci-dessus, lorsque vous instanciez la classe CustomList, vous n'avez pas besoin de spécifier un paramètre de type. Toutes les méthodes ou propriétés de classe de base faisant référence au paramètre de type sont désormais fortement typées dans le type Int32. Par exemple, la méthode List.Add n'acceptera que les arguments de type Int32.

Si la classe de base dont vous héritez contient plusieurs paramètres de type, vous pouvez spécifier des arguments de type pour n'importe quel nombre d'entre eux. La chose importante à retenir est que vous devez soit fournir un argument de type, soit ajouter un paramètre de type correspondant à votre déclaration de classe pour chaque paramètre de type dans le type de base.

L'exemple suivant montre les différentes manières dont vous pouvez hériter d'un type de base avec plusieurs paramètres de type:

**Héritage d'un type de base avec plusieurs paramètres de type**

// Passe tous les paramètres de type de base à la classe dérivée.

classe publique CustomDictionary1 <TKey, TValue>: Dictionary <TKey, TValue> {}

// Fournit un argument pour l'un des paramètres de type de base et passe l'autre à la classe dérivée.

classe publique CustomDictionary2 <TValue>: Dictionary <int, TValue> {}

// Fournit des arguments pour les deux paramètres de type de base.

classe publique CustomDictionary3: Dictionary <int, string> {}

Quel que soit le nombre de paramètres de type (le cas échéant) inclus dans le type de base, vous pouvez ajouter des paramètres de type supplémentaires à vos déclarations de classe dérivées.

L'exemple suivant montre comment ajouter des paramètres de type supplémentaires aux déclarations de classes dérivées:

**Ajout de paramètres de type aux déclarations de classe dérivées**

// Passez le paramètre de type de base à la classe dérivée et ajoutez un paramètre de type supplémentaire.

classe publique CustomCollection1 <T, U>: List <T>

// Fournit un argument pour le paramètre de type de base, mais ajoute un nouveau paramètre de type.

classe publique CustomCollection2 <T>: List <int>

// Hérite d'une classe non générique, mais ajoute un paramètre de type.

classe publique CustomCollection3 <T>: CustomBaseClass

## **Création de méthodes d'extension**

Dans la plupart des cas, si vous souhaitez étendre les fonctionnalités d'une classe, vous utilisez l'héritage pour créer une classe dérivée. Cependant, ce n'est pas toujours possible. De nombreux types intégrés sont scellés pour empêcher l'héritage. Par exemple, vous ne pouvez pas créer une classe qui étend le type System.String.

Au lieu d'utiliser l'héritage pour étendre un type, vous pouvez créer des méthodes d'extension. Lorsque vous créez des méthodes d'extension, vous créez des méthodes que vous pouvez appeler sur un type particulier sans modifier réellement le type sous-jacent. Une méthode d'extension est un type de méthode statique. Pour créer une méthode d'extension, vous créez une méthode statique dans une classe statique. Le premier paramètre de la méthode spécifie le type que vous souhaitez étendre. En faisant précéder le paramètre du mot-clé this, vous indiquez au compilateur que votre méthode est une méthode d'extension de ce type.

L'exemple suivant montre comment créer une méthode d'extension pour le type System.String:

**Créer une méthode d'extension**

namespace FourthExtensionMethods;

{

public static class FourthCoffeeExtensions

{

public static bool ContainsNumbers (this String s)

{

// Utilisez des expressions régulières pour déterminer si la chaîne contient des chiffres numériques.

return Regex.IsMatch (s, @ "\ d");

}

}

}

Pour utiliser une méthode d'extension, vous devez importer explicitement l'espace de noms qui contient votre méthode d'extension à l'aide d'une directive using:

**Intégration d'une méthode d'extension dans la portée**

en utilisant FourthExtensionMethods;

Vous pouvez ensuite appeler la méthode d'extension comme s'il s'agissait d'une méthode d'instance sur le type qu'elle étend:

**Appel d'une méthode d'extension**

Console.WriteLine ("Veuillez saisir du texte contenant des chiffres, puis appuyez sur Entrée");

string text = Console.ReadLine ();

if (text.ContainsNumbers)

{

Console.WriteLine ("Votre texte contient des nombres. Bravo!");

}

autre

{

Console.WriteLine ("Votre texte ne contient pas de chiffres. Veuillez réessayer.");

}

## **Démonstration: refactorisation des fonctionnalités communes dans le laboratoire de classes d'utilisateurs**

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: Refactoring des fonctionnalités communes dans la section User Class Lab sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md>.

# **Atelier: Refactorisation des fonctionnalités communes dans la classe d'utilisateurs**

### **Scénario**

Vous avez remarqué que les classes Étudiant et Enseignant de l'application Notes contiennent des fonctionnalités dupliquées. Pour rendre l'application plus maintenable, vous décidez de refactoriser cette fonctionnalité commune pour supprimer la duplication.

Vous êtes également préoccupé par la sécurité. Les enseignants et les étudiants ont tous besoin d'un mot de passe, mais il est important de maintenir la confidentialité et en même temps de s'assurer que les étudiants (qui sont des enfants) n'ont pas à se souvenir de mots de passe longs et complexes. Vous décidez de mettre en œuvre différentes politiques de mot de passe pour les enseignants et les étudiants; Les mots de passe des enseignants doivent être plus forts et plus difficiles à deviner que les mots de passe des élèves.

De plus, il vous a été demandé de mettre à jour l'application pour limiter le nombre d'élèves pouvant être ajoutés à une classe. Vous décidez d'ajouter du code qui lève une exception personnalisée si un utilisateur tente d'inscrire un étudiant dans une classe qui est déjà à pleine capacité.

### **Objectifs**

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

|  |  |
| --- | --- |
| • | Utilisez l'héritage pour intégrer des fonctionnalités communes dans une classe de base. |
| • | Implémentez le polymorphisme à l'aide d'une méthode abstraite. |
| • | Créez une classe d'exception personnalisée. |

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

Durée estimée: 60 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_MOD05_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_MOD05_LAK.md>.

### **Exercice 1: Création et héritage de la classe de base d'utilisateurs**

##### **Scénario**

Dans cet exercice, vous allez créer une classe de base abstraite appelée User qui contient les propriétés UserName et Password, ainsi que la méthode VerifyPassword commune aux classes Student et Teacher.

Vous allez modifier les définitions des classes Student et Teacher pour hériter de la classe User et supprimer les propriétés UserName et Password et la méthode VerifyPassword de ces classes. Enfin, vous allez créer et exécuter l'application sans apporter d'autres modifications à l'application, puis vérifier qu'elle fonctionne toujours correctement.

**Résultat**: Après avoir terminé cet exercice, vous devez avoir supprimé le code dupliqué des classes Etudiant et Enseignant, et déplacé le code vers une classe de base abstraite appelée Utilisateur.

### **Exercice 2: Implémentation de la complexité des mots de passe à l'aide d'une méthode abstraite**

##### **Scénario**

Dans cet exercice, vous allez ajouter une méthode abstraite appelée SetPassword à la classe User. Dans les classes Enseignant et Etudiant, vous implémenterez la méthode SetPassword. Cette méthode définira le mot de passe de l'utilisateur (un enseignant ou un étudiant). La méthode SetPassword pour un enseignant vérifie que le mot de passe comporte au moins huit caractères et contient au moins deux caractères numériques. La méthode SetPassword pour un étudiant vérifie que le mot de passe comporte au moins six caractères. Si le mot de passe répond à ces exigences, il est défini et la méthode retournera true, sinon elle retournera false. Vous allez ensuite modifier l'accesseur set de la propriété Password dans la classe User pour appeler la méthode SetPassword pour changer le mot de passe de l'utilisateur. Suivant, vous intégrerez cette fonctionnalité dans l'interface utilisateur de l'application pour permettre à un utilisateur de changer son mot de passe. Enfin, vous allez créer et exécuter l'application pour tester la fonctionnalité de mot de passe.

**Résultat**: Après avoir terminé cet exercice, vous devriez avoir implémenté une méthode polymorphe nommée SetPassword qui présente un comportement différent pour les étudiants et les enseignants. Vous aurez également modifié l'application pour permettre aux utilisateurs de changer leurs mots de passe.

### **Exercice 3: Création de l'exception personnalisée ClassFullException**

##### **Scénario**

Dans cet exercice, vous allez créer une nouvelle classe d'exception personnalisée appelée ClassFullException. Vous allez modifier la méthode EnrollInClass de la classe Teacher pour lever cette exception si trop d'étudiants sont ajoutés à la classe d'un enseignant. Vous allez mettre à jour l'application pour intercepter cette exception, puis vous allez créer et exécuter l'application pour tester cette fonctionnalité.

**Résultat**: Après avoir terminé cet exercice, vous devez avoir créé une nouvelle classe d'exception personnalisée et l'utiliser pour signaler un trop grand nombre d'élèves inscrits dans une classe.

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

Dans ce module, vous avez appris à utiliser les méthodes d'héritage et d'extension pour étendre les fonctionnalités des types existants.

### **Question (s) de révision**