**Module 2: Création de méthodes, gestion des exceptions et surveillance des applications**

**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/20483C02.html#P1) |
| **Leçon 1:** | [Créer et appeler des méthodes](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C02.html#P2) |
| **Leçon 2:** | [Création de méthodes surchargées et utilisation de paramètres facultatifs et de sortie](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C02.html#P3) |
| **Lecon 3:** | [Gestion des exceptions](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C02.html#P4) |
| **Leçon 4:** | [Surveillance des applications](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C02.html#P5) |
| **Laboratoire:** | [Extension de la fonctionnalité de l'application d'inscription aux classes](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C02.html#P6) |
|  |  |

# Table des matières

[Contenu: 1](#_Toc67901649)

[Table des matières 2](#_Toc67901650)

[I. Présentation du module 3](#_Toc67901651)

[A. Objectifs 4](#_Toc67901652)

[II. Leçon 1: Créer et appeler des méthodes 4](#_Toc67901653)

[A. Objectifs de la leçon 5](#_Toc67901654)

[B. Qu'est-ce qu'une méthode? 5](#_Toc67901655)

[C. Créer des méthodes 6](#_Toc67901656)

[1. Méthodes de dénomination 7](#_Toc67901657)

[2. Implémentation d'un corps de méthode 7](#_Toc67901658)

[a) Portée de la méthode variable 8](#_Toc67901659)

[3. Spécification des paramètres 8](#_Toc67901660)

[a) Passer des paramètres à une méthode 8](#_Toc67901661)

[b) Définition d'un paramètre à l'aide du mot-clé ref 9](#_Toc67901662)

[4. Spécification d'un type de retour 9](#_Toc67901663)

[a) Renvoyer des données à partir d'une méthode 9](#_Toc67901664)

[b) Ce code ne se compilera pas 10](#_Toc67901665)

[D. Invoquer des méthodes 10](#_Toc67901666)

[a) Appel d'une méthode passant des paramètres 11](#_Toc67901667)

[b) Capture d'une valeur de retour de méthode 12](#_Toc67901668)

[E. Méthodes de débogage 12](#_Toc67901669)

[**Démonstration: création, appel et débogage de méthodes** 14](#_Toc67901670)

[**Démonstration étapes** 14](#_Toc67901671)

[III. Leçon 2: Création de méthodes surchargées et utilisation de paramètres facultatifs et de sortie 14](#_Toc67901672)

[A. Objectifs de la leçon 15](#_Toc67901673)

[B. Création de méthodes surchargées 15](#_Toc67901674)

[a) Méthodes surchargées 16](#_Toc67901675)

[C. Création de méthodes utilisant des paramètres facultatifs 17](#_Toc67901676)

[a) Définition d'une méthode avec des paramètres facultatifs 18](#_Toc67901677)

[b) Définition de paramètre facultative incorrecte 18](#_Toc67901678)

[c) Appel d'une méthode spécifiant uniquement des arguments obligatoires. 19](#_Toc67901679)

[d) Appel d'une méthode spécifiant des arguments obligatoires et facultatifs 19](#_Toc67901680)

[D. Appel d'une méthode à l'aide d'arguments nommés 19](#_Toc67901681)

[a) Utilisation d'arguments nommés 20](#_Toc67901682)

[E. Création de méthodes utilisant des paramètres de sortie 20](#_Toc67901683)

[a) Définition des paramètres de sortie 21](#_Toc67901684)

[b) Appel d'une méthode qui accepte un paramètre de sortie 22](#_Toc67901685)

[c) Appel d'une méthode qui accepte un paramètre de sortie et définit le paramètre en ligne 22](#_Toc67901686)

[IV. Leçon 3: Gestion des exceptions 23](#_Toc67901687)

[A. Objectifs de la leçon 23](#_Toc67901688)

[B. Qu'est-ce qu'une exception? 23](#_Toc67901689)

[1. Comment les exceptions se propagent 24](#_Toc67901690)

[2. Le type d'exception 25](#_Toc67901691)

[C. Gestion des exceptions à l'aide d'un bloc Try / Catch 26](#_Toc67901692)

[a) Syntaxe Try / Catch 27](#_Toc67901693)

[b) Gestion des exceptions NullReferenceException et Exception 28](#_Toc67901694)

[D. Utiliser un bloc Final 28](#_Toc67901695)

[a) Essayer / attraper / enfin bloque 29](#_Toc67901696)

[E. Lancer des exceptions 30](#_Toc67901697)

[a) Créer et lancer une exception 31](#_Toc67901698)

[b) Rejeter une exception 32](#_Toc67901699)

# Présentation du module

Les applications sont souvent constituées d'unités logiques de fonctionnalités qui exécutent des fonctions spécifiques, telles que l'accès aux données ou le déclenchement d'un traitement logique. Visual C # est un langage orienté objet et utilise le concept de méthodes pour encapsuler des unités logiques de fonctionnalité. Une méthode peut être aussi simple ou aussi complexe que vous le souhaitez. Par conséquent, il est important de prendre en compte ce qu'il advient de l'état de votre application lorsqu'une exception se produit dans une méthode.

Dans ce module, vous apprendrez comment créer et utiliser des méthodes et comment gérer les exceptions. Vous apprendrez également à utiliser la journalisation et le traçage pour enregistrer les détails de toutes les exceptions qui se produisent.

## Objectifs

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

|  |  |
| --- | --- |
| • | Créez et invoquez des méthodes. |
| • | Créez des méthodes surchargées et utilisez des paramètres facultatifs. |
| • | Gérez les exceptions. |
| • | Surveillez les applications à l'aide de la journalisation, du traçage et du profilage. |

# Leçon 1: Créer et appeler des méthodes

Chaque application existe pour exécuter un algorithme. Wikipedia décrit un algorithme comme «une spécification sans ambiguïté de la façon de résoudre une classe de problèmes». En termes simples, un algorithme est la description de chaque action nécessaire pour exécuter un processus. Par exemple, l'algorithme pour cliquer sur une icône peut être décrit comme suit:

|  |  |
| --- | --- |
| 1. | Localisez l'icône. |
| 2. | Déplacez le curseur de la souris sur cette icône. |
| 3. | Double-cliquez sur le bouton gauche de la souris. |

Notez que chacune de ces actions peut être fractionnée davantage. Chaque fois que de petites actions sont combinées en une seule action cohésive, un nouveau niveau d'abstraction est ajouté.

Dans les langages orientés objet tels que Visual C #, une méthode est une unité de code qui effectue un travail discret. Cela vous permet de créer de nouveaux niveaux d'abstractions comme bon vous semble et de diviser votre solution en composants logiques gérables.

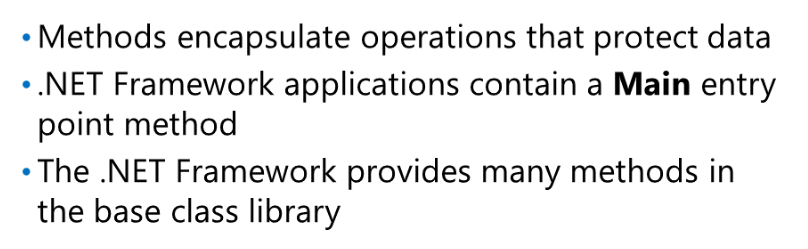
Dans cette leçon, vous apprendrez à créer et à appeler des méthodes.

## Objectifs de la leçon

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

|  |  |
| --- | --- |
| • | Décrivez le but des méthodes. |
| • | Créez des méthodes. |
| • | Invoquez des méthodes. |
| • | Méthodes de débogage. |

## Qu'est-ce qu'une méthode?



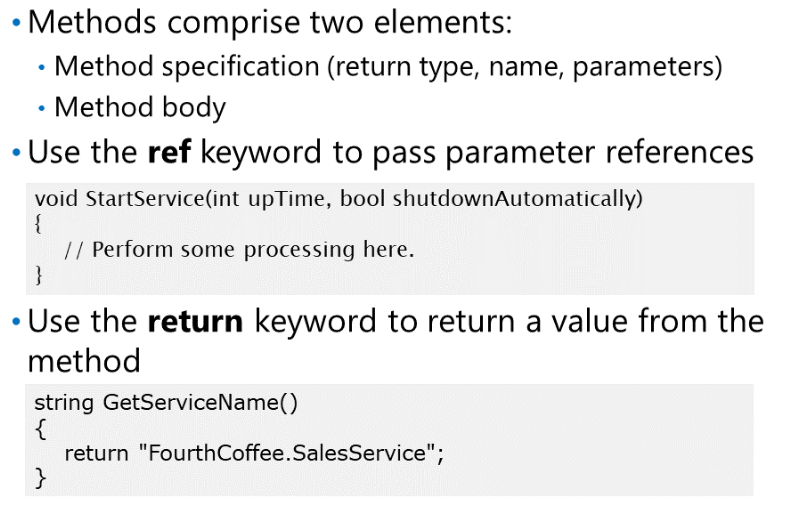
La possibilité de définir et d'appeler des méthodes est un composant fondamental de la programmation orientée objet, car les méthodes vous permettent d'encapsuler des opérations qui protègent les données stockées dans un type.

En règle générale, toute application que vous développez à l'aide de Microsoft .NET Framework et Visual C # aura de nombreuses méthodes, chacune avec un objectif spécifique. Certaines méthodes sont fondamentales pour le fonctionnement d'une application. Par exemple, toutes les applications de bureau Visual C # doivent avoir une méthode appelée Main qui définit le point d'entrée de l'application. Lorsque l'utilisateur exécute une application Visual C #, le Common Language Runtime (CLR) exécute la méthode Main pour cette application.

Les méthodes peuvent être conçues pour un usage interne par un type et, en tant que telles, sont masquées aux autres types. Les méthodes publiques peuvent être conçues pour permettre à d'autres types de demander qu'un objet effectue une action et soient exposés en dehors du type.

Le .NET Framework lui-même est construit à partir de classes qui exposent des méthodes que vous pouvez appeler à partir de vos applications pour interagir avec l'utilisateur et l'ordinateur.

## Créer des méthodes



Un procédé comprend deux éléments:

|  |  |
| --- | --- |
| 1. | La spécification de la méthode. |
| 2. | Le corps de la méthode. |

La **spécification de méthode** définit le **nom de la méthode**, les **paramètres** que la méthode peut prendre, le **type de retour de la méthode** et **l'accessibilité de la méthode**. La combinaison du **nom de la méthode** et de sa liste de **paramètres** est appelée **signature de méthode**. La définition de la valeur de retour d'une méthode n'est pas considérée comme faisant partie de la signature. Chaque méthode d'une classe doit avoir une signature unique.

### Méthodes de dénomination

Un nom de méthode a les mêmes restrictions syntaxiques qu'un nom de variable. Une méthode doit commencer par une lettre ou un trait de soulignement et ne peut contenir que des lettres, des traits de soulignement et des caractères numériques. Visual C # est sensible à la casse, donc une classe peut contenir deux méthodes qui ont le même nom et ne diffèrent que par la casse d'une ou plusieurs lettres, bien que ce ne soit pas une bonne pratique de codage.

Les instructions suivantes sont les meilleures pratiques recommandées lorsque vous choisissez le nom d'une méthode:

|  |  |
| --- | --- |
| • | Utilisez des verbes ou des phrases verbales pour nommer les méthodes. Cela aide les autres développeurs à comprendre la structure de votre code. |
| • | Utilisez UpperCamelCase. |

### Implémentation d'un corps de méthode

Le **corps d'une méthode** est **un bloc de code** implémenté à l'aide de l'une des constructions de programmation Visual C # disponibles. Le **corps** est entouré **d'accolades**.

Vous pouvez définir des **variables à l'intérieur** d'un corps de **méthode**, auquel cas **elles n'existent que pendant l'exécution de la méthode**. Lorsque la **méthode se termine**, **elle n'est plus dans la portée.**

L'exemple de code suivant montre le corps de la méthode StopService, qui contient une variable nommée isServiceRunning. La variable isServiceRunning est uniquement disponible dans le bloc de code StopService. Si vous essayez de faire référence à la variable isServiceRunning en dehors de la portée de la méthode, le compilateur lèvera une erreur de compilation avec le message Le nom «isServiceRunning» n'existe pas dans le contexte actuel.

#### Portée de la méthode variable

void StopService ()

{

var isServiceRunning = FourthCoffeeServices.Status;

...

}

### Spécification des paramètres

Les **paramètres** sont des **variables locales** qui sont créées lorsque la méthode s'exécute et sont remplies avec des valeurs spécifiées lors de l'appel de la méthode. Toutes les méthodes doivent avoir une liste de paramètres. Vous spécifiez les paramètres entre parenthèses après le nom de la méthode. Chaque paramètre est séparé par une virgule. Si une méthode ne prend aucun paramètre, vous spécifiez une liste de paramètres vide.

Pour chaque paramètre, vous spécifiez le type et le nom. Par convention, les **paramètres** sont nommés à l'aide de **lowerCamelCase.**

L'exemple de code suivant montre une méthode qui accepte un paramètre int et un paramètre booléen.

#### Passer des paramètres à une méthode

void StartService (int upTime, bool shutdownAutomatically)

{

// Effectuez un traitement ici.

}

Lors de la définition des paramètres acceptés par une méthode, vous pouvez également préfixer la **définition de paramètre** avec le **mot-clé ref**. En utilisant le mot-clé ref, vous indiquez au **CLR de transmettre une référence au paramètre** et pas seulement la valeur du paramètre. Vous devez initialiser le paramètre ref, et **toute modification du paramètre à l'intérieur du corps de la méthode sera alors reflétée dans la variable sous-jacente de la méthode appelante.**

L'exemple de code suivant montre comment définir un paramètre à l'aide du mot clé ref.

#### Définition d'un paramètre à l'aide du mot-clé ref

void StopAllServices (ref int serviceCount)

{

serviceCount = FourthCoffeeServices.ActiveServiceCount;

}

**Lecture supplémentaire:**Pour plus d'informations sur le mot clé ref, consultez la page ref (référence C #) à l'adresse<http://go.microsoft.com/fwlink/?LinkID=267782>.

### Spécification d'un type de retour

**Toutes les méthodes doivent avoir un type de retour**. Une **méthode qui ne renvoie pas de valeur** a le type de retour **void**. Vous spécifiez le type de retour avant le nom de la méthode lorsque vous définissez une méthode. Lorsque vous déclarez une méthode qui renvoie des données, vous devez inclure une instruction return dans le bloc de méthodes.

L'exemple de code suivant montre comment renvoyer une chaîne à partir d'une méthode.

#### Renvoyer des données à partir d'une méthode

chaîne GetServiceName ()

{

return "FourthCoffee.SalesService";

}

L'expression spécifiée par l'instruction return doit avoir le même type que la méthode. Lorsque l'instruction return s'exécute, cette expression est évaluée et renvoyée à l'instruction qui a appelé la méthode. La méthode se termine alors, de sorte que **toute autre instruction qui se produit après l'exécution d'une instruction return ne sera pas exécutée**.

Cependant, chaque chemin d'exécution de la méthode doit éventuellement appeler le mot-clé return (ou lever une exception). Le compilateur produira une erreur s'il est possible d'exécuter la méthode sans atteindre une instruction return.

L'exemple de code suivant générera une erreur du compilateur car il est possible d'exécuter la méthode sans renvoyer de valeur.

#### Ce code ne se compilera pas

// Erreur CS0161 'GetServiceName ()': tous les chemins de code ne renvoient pas une valeur

string GetServiceName (langage de chaîne)

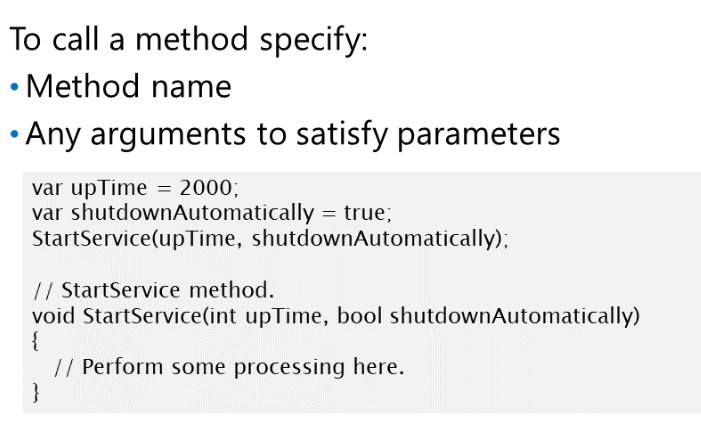
{

Si (langue == "fr")

return "FourthCoffee.SalesService";

}

## Invoquer des méthodes



Vous appelez une méthode pour exécuter le code de cette méthode à partir d'une partie de votre application. Vous n'avez pas besoin de comprendre comment fonctionne le code d'une méthode. Vous n'aurez peut-être même pas accès au code s'il se trouve dans une classe d'un assembly dont vous n'avez pas la source, telle que la bibliothèque de classes .NET Framework.

Pour **appeler une méthode**, vous spécifiez **le nom de la méthode et fournissez tous les arguments** qui **correspondent aux paramètres de méthode entre crochets.**

L'exemple de code suivant montre comment appeler la méthode StartService, en passant des variables int et booléennes pour satisfaire les exigences de paramètre de la signature de la méthode.

#### Appel d'une méthode passant des paramètres

var upTime = 2000;

var shutdownAutomatically = true;

StartService (upTime, shutdownAutomatically);

// Méthode StartService.

void StartService (int upTime, bool shutdownAutomatically)

{

// Effectuez un traitement ici.

}

Si la méthode renvoie une valeur, vous spécifiez comment gérer cette valeur, généralement en l'affectant à une variable du même type, dans votre code d'appel.

L'exemple de code suivant montre comment capturer la valeur de retour de la méthode GetServiceName dans une variable nommée serviceName.

#### Capture d'une valeur de retour de méthode

var serviceName = GetServiceName ();

chaîne GetServiceName ()

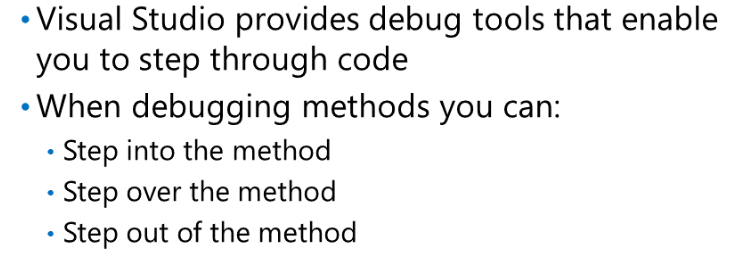
{

return "FourthCoffee.SalesService";

}

**Lecture supplémentaire:**Pour plus d'informations sur les méthodes, consultez la page Méthodes (Guide de programmation C #) à l'adresse<http://go.microsoft.com/fwlink/?LinkID=267774>.

## Méthodes de débogage



Lorsque vous déboguez votre application, vous pouvez parcourir le code une instruction à la fois. Il s'agit d'une fonctionnalité extrêmement utile car elle vous permet de tester la logique selon laquelle votre application utilise une étape à la fois.

Visual Studio fournit un certain nombre d'outils de débogage qui vous permettent de parcourir le code exactement comme vous le souhaitez. Par exemple, vous pouvez parcourir chaque ligne de chaque méthode exécutée ou ignorer les instructions d'une méthode dont vous savez qu'elles fonctionnent correctement. Vous pouvez également parcourir complètement le code, empêchant l'exécution de certaines instructions.

Lors du débogage de méthodes, vous pouvez utiliser les trois fonctionnalités de débogage suivantes pour contrôler si vous passez, entrez ou sortez d'une méthode:

|  |  |
| --- | --- |
| • | La fonction Pas à pas exécute l'instruction à la position d'exécution actuelle. Si l'instruction est un appel de méthode, la position d'exécution actuelle se déplacera vers le code à l'intérieur de la méthode. Une fois que vous êtes entré dans une méthode, vous pouvez continuer à exécuter des instructions à l'intérieur de la méthode, une ligne à la fois. Vous pouvez également utiliser le bouton Pas à pas pour démarrer une application en mode débogage. Si vous faites cela, l'application entrera en mode pause dès son démarrage. |
| • | La fonction Step Over exécute l'instruction à la position d'exécution actuelle. Cependant, cette fonctionnalité n'entre pas dans le code à l'intérieur d'une méthode. Au lieu de cela, le code à l'intérieur de la méthode s'exécute et la position d'exécution se déplace vers l'instruction après l'appel de la méthode. L'exception à cela est si le code de la méthode ou de la propriété contient un point d'arrêt. Si tel est le cas, l'exécution se poursuivra jusqu'au point d'arrêt. L'utilisation de Step Over lorsque l'application est fermée la démarre également en mode débogage et s'arrête sur la première ligne. |
| • | La fonction Step Out vous permet d'exécuter le code restant dans une méthode. L'exécution se poursuivra avec l'instruction qui a appelé la méthode, puis s'arrêtera à ce stade. |

**Lecture supplémentaire:**Pour plus d'informations sur la progression dans le code, consultez le didacticiel: apprendre à déboguer à l'aide de la page Visual Studio à l'adresse<https://aka.ms/moc-20483c-m2-pg1>.

**Démonstration: création, appel et débogage de méthodes**

Dans cette démonstration, vous allez créer une méthode, appeler la méthode, puis déboguer la méthode.

**Démonstration étapes**

Vous trouverez les étapes dans la section «Démonstration: création, appel et débogage de méthodes» sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_DEMO.md>.

# Leçon 2: Création de méthodes surchargées et utilisation de paramètres facultatifs et de sortie

Vous avez vu que vous pouvez définir une méthode qui accepte un nombre fixe de paramètres. Cependant, vous pouvez parfois écrire **une méthode** **générique (**En [programmation](https://fr.wikipedia.org/wiki/Programmation_informatique), la **généricité** (ou **programmation générique**), consiste à définir des [algorithmes](https://fr.wikipedia.org/wiki/Algorithme) identiques opérant sur des [données](https://fr.wikipedia.org/wiki/Donn%C3%A9e_(informatique)) de [types](https://fr.wikipedia.org/wiki/Type_(informatique)) différents**)** qui nécessite différents ensembles de paramètres en fonction du contexte dans lequel elle est utilisée. Vous pouvez créer des méthodes surchargées avec des signatures uniques pour répondre à ce besoin. Dans d'autres scénarios, vous souhaiterez peut-être définir une méthode qui a un nombre fixe de paramètres, mais qui permet à une application de spécifier des arguments uniquement pour les paramètres dont elle a besoin. Vous pouvez le faire en définissant une méthode qui prend des paramètres facultatifs, puis en utilisant des arguments nommés pour satisfaire les paramètres par leur nom.

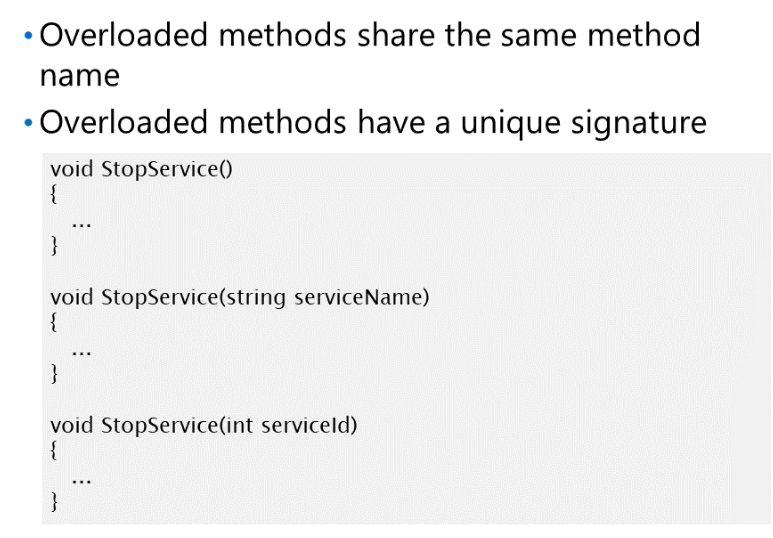
Dans cette leçon, vous apprendrez à créer des méthodes surchargées, à définir et à utiliser des paramètres facultatifs, des arguments nommés et des paramètres de sortie.

## Objectifs de la leçon

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

|  |  |
| --- | --- |
| • | Créez une méthode surchargée. |
| • | Utilisez des paramètres facultatifs. |
| • | Utilisez des arguments nommés. |
| • | Définissez les paramètres de sortie. |

## Création de méthodes surchargées



Lorsque vous définissez une méthode, vous pouvez vous rendre compte qu'elle nécessite différents ensembles d'informations dans différentes circonstances. Vous pouvez définir des méthodes surchargées pour créer plusieurs méthodes avec la même fonctionnalité qui acceptent différents paramètres en fonction du contexte dans lequel elles sont appelées.

Les **méthodes surchargées** portent le **même nom** les unes que les autres pour souligner leur intention commune. Cependant, chaque méthode surchargée doit avoir une **signature unique**, pour la différencier des autres versions surchargées de la méthode dans la classe.

La **signature** d'une méthode comprend son **nom** et sa **liste de paramètres**. **Le type de retour ne fait pas partie de la signature**. Par conséquent, vous ne pouvez pas définir des méthodes surchargées qui diffèrent uniquement par leur type de retour.

L'exemple de code suivant montre trois versions de la méthode StopService, toutes avec une signature unique.

#### Méthodes surchargées

void StopService ()

{

...

}

void StopService (chaîne serviceName)

{

...

}

void StopService (int serviceId)

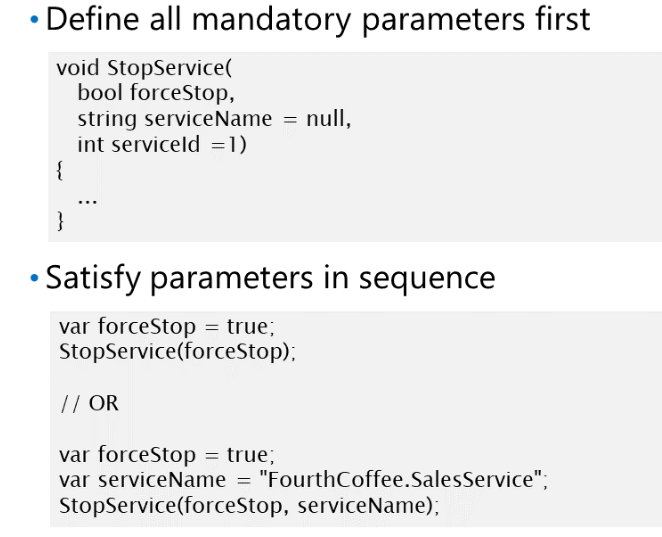
{

...

}

Lorsque vous appelez la méthode StopService, vous avez le choix de la version surchargée que vous utilisez. Vous fournissez simplement les arguments pertinents pour satisfaire une surcharge particulière, puis le compilateur détermine la version à invoquer en fonction des arguments que vous avez transmis.

## Création de méthodes utilisant des paramètres facultatifs



En définissant des méthodes surchargées, vous pouvez implémenter différentes versions d'une méthode prenant différents paramètres. Lorsque vous créez une application qui utilise des méthodes surchargées, le compilateur détermine quelle instance spécifique de chaque méthode il doit utiliser pour satisfaire chaque appel de méthode.

Il existe d'autres langages et technologies que les développeurs peuvent utiliser pour créer des applications et des composants qui ne respectent pas ces règles. Une fonctionnalité clé de Visual C # est la possibilité d'interagir avec des applications et des composants écrits à l'aide d'autres technologies. L'une des principales technologies utilisées par Windows est le modèle d'objet composant (COM). COM ne prend pas en charge les méthodes surchargées, mais utilise à la place des méthodes qui peuvent accepter des paramètres facultatifs. Pour faciliter l'incorporation de bibliothèques et de composants COM dans une solution Visual C #, Visual C # prend également en charge les paramètres facultatifs.

Les paramètres facultatifs sont également utiles dans d'autres situations. Ils fournissent une solution compacte et simple lorsqu'il n'est pas possible d'utiliser la surcharge car les types de paramètres ne varient pas suffisamment pour permettre au compilateur de distinguer les implémentations.

L'exemple de code suivant montre comment définir une méthode qui accepte un paramètre obligatoire et deux paramètres facultatifs.

#### Définition d'une méthode avec des paramètres facultatifs

void StopService (bool forceStop, string serviceName = null, int serviceId = 1)

{

...

}

Lors de la définition d'une méthode qui accepte des paramètres facultatifs, vous devez spécifier tous les paramètres obligatoires avant tout paramètre facultatif.

L'exemple de code suivant montre une définition de méthode qui utilise des paramètres facultatifs qui lève une erreur de compilation.

#### Définition de paramètre facultative incorrecte

void StopService (string serviceName = null, bool forceStop, int serviceId = 1)

{

...

}

Vous pouvez appeler une méthode qui prend des paramètres facultatifs de la même manière que vous appelez n'importe quelle autre méthode. Vous spécifiez le nom de la méthode et fournissez tous les arguments nécessaires. La différence avec les méthodes qui acceptent des paramètres facultatifs est que vous pouvez omettre les arguments correspondants et que la méthode utilisera la valeur par défaut lors de l'exécution de la méthode.

L'exemple de code suivant montre comment appeler la méthode StopService, en transmettant uniquement un argument pour le paramètre obligatoire forceStop.

#### Appel d'une méthode spécifiant uniquement des arguments obligatoires.

var forceStop = vrai;

StopService (forceStop);

L'exemple de code suivant montre comment appeler la méthode StopService, en passant un argument pour le paramètre obligatoire forceStop et un argument pour le paramètre serviceName.

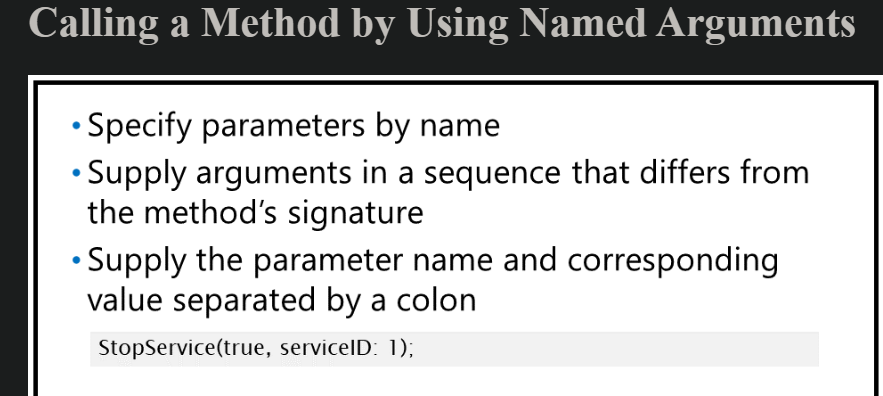
#### Appel d'une méthode spécifiant des arguments obligatoires et facultatifs

var forceStop = vrai;

var serviceName = "FourthCoffee.SalesService";

StopService (forceStop, serviceName);

## Appel d'une méthode à l'aide d'arguments nommés



Traditionnellement, lors de l'appel d'une méthode, l'ordre et la position des arguments dans l'appel de méthode correspondent à l'ordre des paramètres dans la signature de la méthode. Si les arguments ne sont pas alignés et que les types ne correspondent pas, vous recevez une erreur de compilation.

Dans Visual C #, vous pouvez spécifier des paramètres par nom et par conséquent fournir des arguments dans une séquence qui diffère de celle définie dans la signature de la méthode. Pour utiliser des arguments nommés, vous fournissez le nom du paramètre et la valeur correspondante séparés par deux points.

L'exemple de code suivant montre comment appeler la méthode StopService à l'aide d'arguments nommés pour transmettre le paramètre serviceID.

#### Utilisation d'arguments nommés

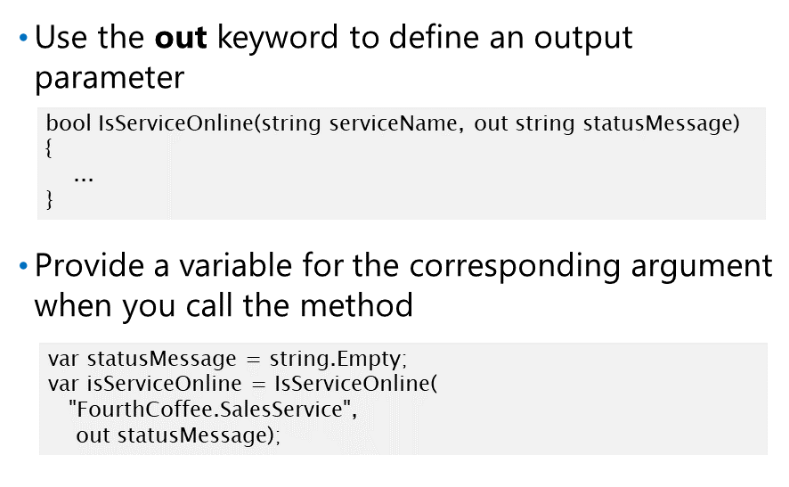
StopService (vrai, serviceID: 1);

Lorsque vous utilisez des arguments nommés conjointement avec des paramètres facultatifs, vous pouvez facilement omettre des paramètres. Tous les paramètres facultatifs recevront leur valeur par défaut. Cependant, si vous omettez des paramètres obligatoires, votre code ne sera pas compilé.

Vous pouvez mélanger des arguments positionnels et nommés. Cependant, vous devez spécifier tous les arguments de position avant les arguments nommés.

**Lecture supplémentaire:**Pour plus d'informations sur l'utilisation des arguments nommés, consultez la page Arguments nommés et facultatifs (Guide de programmation C #) à l'adresse<http://go.microsoft.com/fwlink/?LinkID=267784>.

## Création de méthodes utilisant des paramètres de sortie



Une méthode peut renvoyer une valeur au code qui l'appelle à l'aide d'une instruction return. Si vous devez renvoyer plusieurs valeurs au code appelant, vous pouvez utiliser des paramètres de sortie pour renvoyer des données supplémentaires à partir de la méthode. Lorsque vous ajoutez un paramètre de sortie à une méthode, le corps de la méthode est censé affecter une valeur à ce paramètre. Une fois la méthode terminée, la valeur du paramètre de sortie est affectée à une variable spécifiée comme argument correspondant dans l'appel de méthode.

Pour définir un paramètre de sortie, vous préfixez le paramètre dans la signature de méthode avec le mot clé out.

L'exemple de code suivant montre comment définir une méthode qui utilise des paramètres de sortie

#### Définition des paramètres de sortie

bool IsServiceOnline (string serviceName, out string statusMessage)

{

var isOnline = FourthCoffeeServices.GetStatus (serviceName);

if (isOnline)

{

statusMessage = "Les services sont en cours d'exécution.";

}

autre

{

statusMessage = "Les services sont actuellement arrêtés.";

}

return isOnline;

}

Une méthode peut avoir autant de paramètres de sortie que nécessaire. Lorsque vous déclarez un paramètre de sortie, vous devez attribuer une valeur au paramètre avant le retour de la méthode, sinon le code ne sera pas compilé.

Pour utiliser un paramètre de sortie, vous devez fournir une variable pour l'argument correspondant lorsque vous appelez la méthode et préfixez cet argument avec le mot-clé out. Si vous essayez de spécifier un argument qui n'est pas une variable ou si vous omettez le mot-clé out, votre code ne sera pas compilé.

L'exemple de code suivant montre comment appeler une méthode qui accepte un paramètre de sortie.

#### Appel d'une méthode qui accepte un paramètre de sortie

var statusMessage = string.Empty;

var isServiceOnline = IsServiceOnline ("FourthCoffee.SalesService", out statusMessage);

Les nouvelles versions de Visual C # vous permettent de définir la variable en ligne avec le mot clé out. Cela permet de supprimer la première ligne définissant la variable vide. Cela fonctionne également avec les paramètres ref.

L'exemple de code suivant montre comment appeler une méthode qui accepte un paramètre de sortie et définir la variable en ligne

#### Appel d'une méthode qui accepte un paramètre de sortie et définit le paramètre en ligne

var isServiceOnline = IsServiceOnline ("FourthCoffee.SalesService", out var statusMessage);

**Lecture supplémentaire:**Pour plus d'informations sur les paramètres de sortie, consultez la page modificateur de paramètre de sortie (référence C #) à l'adresse<http://go.microsoft.com/fwlink/?LinkID=267785>.

# Leçon 3: Gestion des exceptions

La gestion des exceptions est un concept important pour garantir une bonne expérience utilisateur et limiter les pertes de données. Les applications doivent être conçues en tenant compte de la gestion des exceptions.

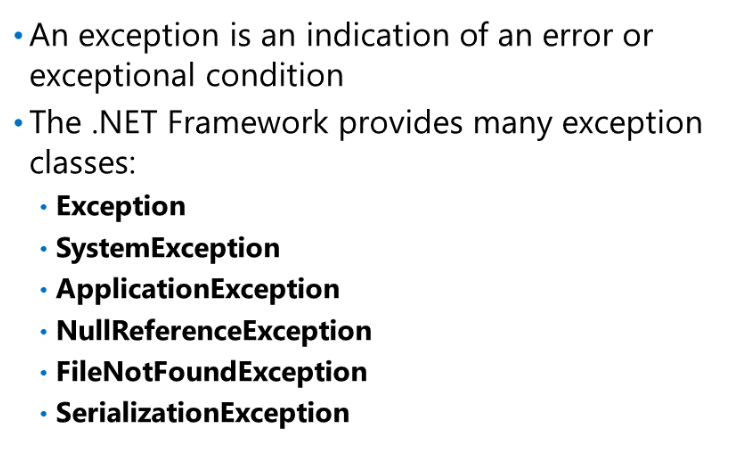
Dans cette leçon, vous apprendrez comment implémenter une gestion efficace des exceptions dans vos applications et comment vous pouvez utiliser des exceptions dans vos méthodes pour indiquer avec élégance une condition d'erreur au code qui appelle vos méthodes.

## Objectifs de la leçon

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

|  |  |
| --- | --- |
| • | Décrivez le but d'une exception. |
| • | Gérez les exceptions à l'aide d'un bloc try / catch. |
| • | Utilisez un bloc finally pour exécuter du code après une exception. |
| • | Lancez une exception. |

## Qu'est-ce qu'une exception?



Beaucoup de choses peuvent mal tourner lors de l'exécution d'une application. Certaines erreurs peuvent survenir en raison de failles dans la logique de l'application, mais d'autres peuvent être dues à des conditions indépendantes de la volonté de votre application. Par exemple, votre application ne peut pas garantir qu'un fichier existe sur le système de fichiers ou qu'une base de données requise est en ligne. Lorsque vous concevez une application, vous devez déterminer comment vous assurer que votre application peut récupérer correctement lorsque de tels problèmes surviennent. Il est courant de vérifier simplement les valeurs de retour des méthodes pour s'assurer qu'elles se sont exécutées correctement, cependant, cette méthodologie n'est pas toujours suffisante pour gérer toutes les erreurs qui peuvent survenir car:

|  |  |
| --- | --- |
| • | Toutes les méthodes ne renvoient pas une valeur. |
| • | Vous devez savoir pourquoi l'appel de méthode a échoué, pas seulement qu'il a échoué. |
| • | Les erreurs inattendues telles que le manque de mémoire ne peuvent pas être traitées de cette manière. |

Traditionnellement, les applications utilisaient le concept d'objet d'erreur global. Lorsqu'un morceau de code provoquait une erreur, il définissait les données de cet objet pour indiquer la cause de l'erreur, puis retournait à l'appelant. Il incombait au code appelant d'examiner l'objet d'erreur et de déterminer comment le gérer. Cependant, cette approche n'est pas robuste, car il est trop facile pour un programmeur d'oublier de gérer les erreurs de manière appropriée.

### Comment les exceptions se propagent

Le .NET Framework utilise des exceptions pour aider à surmonter ces problèmes. Une exception est l'indication d'une erreur ou d'une condition exceptionnelle. Une méthode peut lever une exception lorsqu'elle détecte qu'un événement inattendu s'est produit, par exemple, l'application tente d'ouvrir un fichier qui n'existe pas.

Lorsqu'une méthode lève une exception, le code appelant doit être préparé pour détecter et gérer cette exception. Si le code appelant ne détecte pas l'exception, le code est abandonné et l'exception est automatiquement propagée au code qui a appelé le code appelant. Ce processus se poursuit jusqu'à ce qu'une section de code prenne la responsabilité de gérer l'exception. L'exécution se poursuit dans cette section de code une fois la logique de gestion des exceptions terminée. Si aucun code ne gère l'exception, le processus se terminera et affichera un message à l'utilisateur.

### Le type d'exception

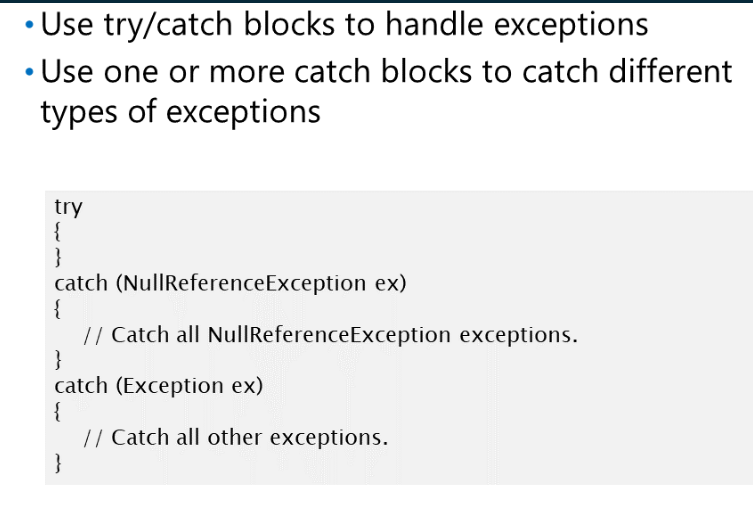
Lorsqu'une exception se produit, il est utile d'inclure des informations sur la cause d'origine afin que la méthode qui gère l'exception puisse prendre l'action corrective appropriée. Dans le .NET Framework, les exceptions sont basées sur la classe Exception, qui contient des informations sur l'exception. Lorsqu'une méthode lève une exception, elle crée un objet Exception et peut le renseigner avec des informations sur la cause de l'erreur. L'objet Exception est ensuite passé au code qui gère l'exception.

Le tableau suivant décrit certaines des classes d'exception fournies par le .NET Framework.

| **Classe d'exception** | **Espace de noms** | **Description** |
| --- | --- | --- |
| **Exception** | **Système** | Représente toute exception déclenchée lors de l'exécution d'une application. |
| **SystemException** | **Système** | Représente toutes les exceptions levées par le CLR. La classe SystemException est la classe de base pour toutes les classes d'exception dans l'espace de noms System. |
| **ApplicationException** | **Système** | Représente toutes les exceptions non fatales déclenchées par les applications et non par le CLR. |
| **NullReferenceException** | **Système** | Représente une exception provoquée lors de la tentative d'utilisation d'un objet nul. |
| **FileNotFoundException** | **System.IO** | Représente une exception provoquée lorsqu'un fichier n'existe pas. |
| **SerializationException** | **System.Runtime.Serialization** | Représente une exception qui se produit pendant le processus de sérialisation ou de désérialisation. |

**Lecture supplémentaire:**Pour plus d'informations sur la classe Exception, consultez la page Classe d'exception à l'adresse<https://aka.ms/moc-20483c-m2-pg2>.

## Gestion des exceptions à l'aide d'un bloc Try / Catch



Le bloc try / catch est la construction de programmation clé qui vous permet d'implémenter la gestion structurée des exceptions (SEH) dans vos applications. Vous encapsulez le code qui peut échouer et provoquer une exception dans un bloc try et ajoutez un ou plusieurs blocs catch pour gérer les exceptions qui peuvent se produire.

L'exemple de code suivant montre la syntaxe pour définir un bloc try / catch.

#### Syntaxe Try / Catch

try

{

// Essayez le bloc.

}

catch ([spécification de capture 1])

{

// Attrapez le bloc 1.

}

catch ([spécification catch n])

{

// Bloc de capture n.

}

Les instructions placées entre accolades dans le bloc try peuvent être des instructions Visual C # et peuvent appeler des méthodes dans d'autres objets. Si l'une de ces instructions entraîne la levée d'une exception, l'exécution passe au bloc catch approprié. La spécification catch pour chaque bloc détermine les exceptions qu'il interceptera et la variable, le cas échéant, dans laquelle stocker l'exception. Vous pouvez spécifier des blocs catch pour différents types d'exceptions. Il est recommandé d'inclure un bloc catch pour le type général Exception à la fin des blocs catch pour intercepter toutes les exceptions qui n'auraient pas été gérées autrement.

Dans l'exemple de code suivant, si le code du bloc try provoque une exception NullReferenceException, le code du bloc catch correspondant s'exécute. Si un autre type d'exception se produit, le code du bloc catch pour le type Exception s'exécute.

#### Gestion des exceptions NullReferenceException et Exception

try

{

}

catch (NullReferenceException ex)

{

// Attrape toutes les exceptions NullReferenceException.

}

catch (Exception ex)

{

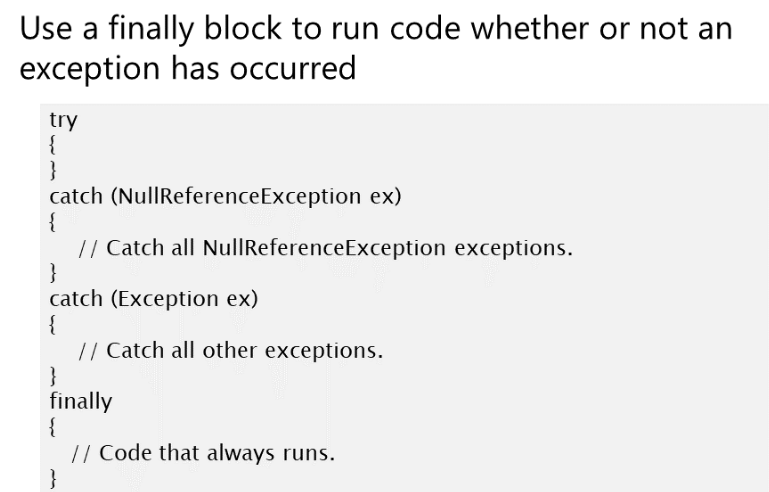
// Attrape toutes les autres exceptions.

}

Lorsque vous définissez plusieurs blocs catch, vous devez vous assurer de les placer dans le bon ordre. Lorsqu'une exception est levée, le CLR tente de faire correspondre l'exception à chaque bloc catch à son tour. Vous devez placer des blocs catch plus spécifiques avant des blocs catch moins spécifiques, sinon votre code ne sera pas compilé.

**Lecture supplémentaire:**Pour plus d'informations sur les blocs try / catch, consultez la page try-catch (référence C #) à l'adresse<https://aka.ms/moc-20483c-m2-pg3>.

## Utiliser un bloc Final



Certaines méthodes peuvent contenir du code critique qui doit toujours être exécuté, même si une exception non gérée se produit. Par exemple, une méthode peut avoir besoin de s'assurer qu'elle ferme un fichier dans lequel elle était en train d'écrire ou libère d'autres ressources avant de s'arrêter. Un bloc finally vous permet de gérer cette situation.

Vous spécifiez un bloc finally après tous les gestionnaires catch dans un bloc try / catch. Il spécifie le code qui doit être exécuté lorsque le bloc se termine, indépendamment du fait que des exceptions, gérées ou non, se produisent. Si une exception est interceptée et gérée, le gestionnaire d'exceptions dans le bloc catch s'exécutera avant le bloc finally.

Vous pouvez également ajouter un bloc finally au code qui n'a pas de blocs catch. Dans ce cas, toutes les exceptions ne sont pas gérées, mais le bloc finally s'exécutera toujours.

L'exemple de code suivant montre comment implémenter un bloc try / catch / finally.

#### Essayer / attraper / enfin bloque

try

{

}

catch (NullReferenceException ex)

{

// Attrape toutes les exceptions NullReferenceException.

}

catch (Exception ex)

{

// Attrape toutes les autres exceptions.

}

finally

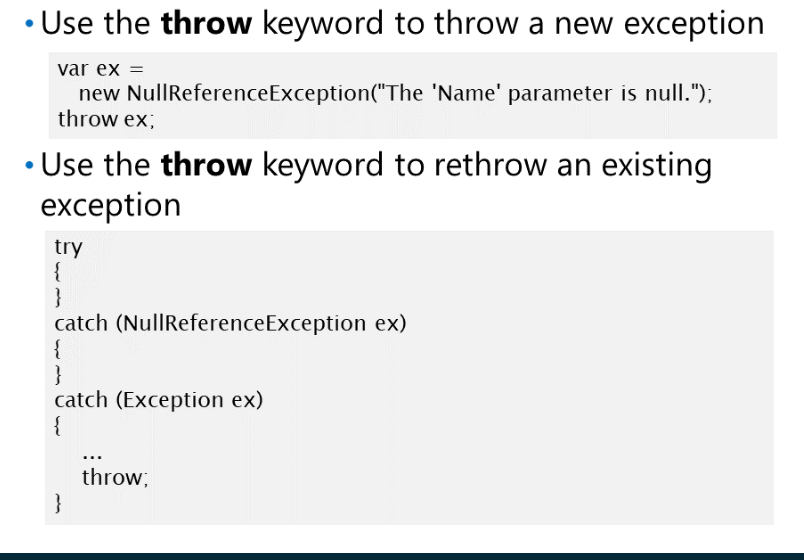
{

// Code qui s'exécute toujours.

}

**Lecture supplémentaire:**Pour plus d'informations sur les blocs try / catch / finally, consultez la page try-catch-finally (référence C #) à l'adresse<https://aka.ms/moc-20483c-m2-pg4>.

## Lancer des exceptions



Vous pouvez créer une instance d'une classe d'exception dans votre code et lever l'exception pour indiquer qu'une exception s'est produite. Lorsque vous lancez une exception, l'exécution du bloc de code actuel se termine et le CLR passe le contrôle au premier gestionnaire d'exceptions disponible qui intercepte l'exception.

Pour lever une exception, vous utilisez le mot-clé throw et spécifiez l'objet d'exception à lever.

L'exemple de code suivant montre comment créer une instance de la classe NullReferenceException, puis lancer l'objet ex.

#### Créer et lancer une exception

var ex = new NullReferenceException ("Le paramètre 'Name' est nul.");

throw ex;

Une stratégie courante consiste à ce qu'une méthode ou un bloc de code intercepte toutes les exceptions et tente de les gérer. Si le bloc catch pour une exception ne peut pas résoudre l'erreur, il peut renvoyer l'exception pour la propager à l'appelant.

L'exemple de code suivant montre comment renvoyer une exception qui a été interceptée dans un bloc catch.

#### Rejeter une exception

try

{

}

catch (NullReferenceException ex)

{

// Attrape toutes les exceptions NullReferenceException.

}

catch (Exception ex)

{

// Tentative de gestion de l'exception

...

// Si ce gestionnaire catch ne peut pas résoudre l'exception,

// jette-le au code appelant

lancer;

}