Contents

- Documentation C#
- Prise en main
  - Introduction au langage C# et à .NET
  - Présentation de C#
    - Introduction
    - Types
    - Blocs de construction de programme
    - Principales zones de langue
- Didacticiels
  - Vue d'ensemble
  - Introduction à la programmation avec C#
    - Choisir votre première leçon
    - Hello World
    - Nombres en C#
    - Branches et boucles
    - Lister les collections
    - Travailler dans votre environnement local
      - Configurer votre environnement
      - Nombres en C#
      - Branches et boucles
      - Lister les collections
      - Introduction aux classes
      - Programmation orientée objet
  - Explorer les types d'enregistrement
  - Explorer les instructions de niveau supérieur
  - Explorer les modèles dans les objets
  - Explorer l'interpolation de chaîne - interactif
  - Explorer l'interpolation de chaîne - dans votre environnement
  - Scénarios avancés d'interpolation de chaîne

Mettre à jour de manière sécurisée des interfaces avec les méthodes d'interface par défaut
Créer une fonctionnalité mixin avec les méthodes d'interface par défaut
Explorer des index et des plages
Utiliser des types référence nullable
Mettre à niveau une application avec des types référence nullable
Générer et consommer des flux asynchrones
Créer des algorithmes pilotés par les données avec des critères spéciaux
Application console
Client REST
Héritage dans C# et .NET
Utiliser LINQ
Utiliser des attributs
Nouveautés de C#
C# 9.0
C# 8.0
C# 7.0-7.3
Changements importants dans le compilateur
Historique des versions de C#
Relations entre le langage et le framework
Considérations relatives à la version et la mise à jour
Concepts C#
Système de type C#
Types références Nullables
Choisir une stratégie pour activer les types référence Nullable
Espaces de noms
Types de base
Classes
Déconstruction de tuples et d'autres types
Interfaces
Méthodes
Propriétés
Indexeurs

Éléments ignorés
Génériques
Iterators
Délégués et événements
  Présentation des délégués
  System.Delegate et mot clé delegate
  Délégués fortement typés
  Modèles courants pour les délégués
  Introduction aux événements
  Modèles d'événement .NET standard
  Modèle d'événement .NET mis à jour
  Différenciation des délégués et des événements
LINQ (Language Integrated Query)
  Vue d'ensemble de LINQ
  Concepts de base des expressions de requête
  LINQ en C#
  Écrire des requêtes LINQ en C#
  Interroger une collection d'objets
  Retourner une requête à partir d'une méthode
  Stocker les résultats d'une requête dans la mémoire
  Regrouper les résultats d'une requête
  Créer un groupe imbriqué
  Effectuer une sous-requête sur une opération de regroupement
  Regrouper des résultats par clés contiguës
  Spécifier dynamiquement des filtres de prédicat au moment de l'exécution
  Effectuer des jointures internes
  Effectuer des jointures groupées
  Effectuer des jointures externes gauches
  Classer les résultats d'une clause join
  Effectuer des jointures à l'aide de clés composites
  Effectuer des opérations de jointure personnalisées
  Gérer des valeurs Null dans des expressions de requête

Gérer des exceptions dans des expressions de requête
Programmation asynchrone
Critères spéciaux
Écrire du code sécurisé et efficace
Arborescences de l'expression
Introduction aux arborescences d'expressions
Explication des arborescences d'expressions
Types de frameworks prenant en charge les arborescences d'expressions
Exécution des expressions
Interprétation des expressions
Création des expressions
Traduction des expressions
Récapitulatif
Interopérabilité native
Documentation de votre code
Gestion de version
Articles de procédure C#
Index de l'article
Fractionner des chaînes en sous-chaînes
Concaténer des chaînes
Rechercher des chaînes
Modifier le contenu des chaînes
Comparer des chaînes
Effectuer des casts sécurisés avec les critères spéciaux et les opérateurs is/as
Kit SDK .NET Compiler Platform (API Roslyn)
Vue d'ensemble du kit SDK .NET Compiler Platform (API Roslyn)
Comprendre le modèle de l'API du compilateur
Utiliser la syntaxe
Utiliser la sémantique
Utiliser un espace de travail
Explorer le code avec le visualiseur de syntaxe
Démarrages rapides

- Analyse de la syntaxe
- Analyse sémantique
- Transformation de syntaxe
- Didacticiels
  - Créer votre premier analyseur et correcteur de code
- Guide de programmation C#
  - Vue d'ensemble
  - À l'intérieur d'un programme C#
    - Que contient un programme C# ?
    - Hello World -- Votre premier programme
    - Structure générale d'un programme C#
    - Noms d'identificateurs
    - Conventions de codage C#
  - Main() et arguments de ligne de commande
    - Vue d'ensemble
    - Arguments de ligne de commande
    - Comment afficher les arguments de ligne de commande
    - Valeurs de retour Main()
  - Concepts de programmation
    - Vue d'ensemble
    - Programmation asynchrone
      - Vue d'ensemble
      - Modèle de programmation asynchrone des tâches
      - Types de retour asynchrones
      - Annuler des tâches
        - Annuler une liste de tâches
        - Annuler des tâches après une période de temps
      - Traiter les tâches asynchrones terminées
      - Accès asynchrone aux fichiers
    - Attributs
      - Vue d'ensemble
      - Création d'attributs personnalisés

Accès à des attributs à l'aide de la réflexion
Guide pratique pour créer une union C/C++ en utilisant des attributs
Collections
Covariance et contravariance
Vue d'ensemble
Variance dans les interfaces génériques
Créer des interfaces génériques de type variant
Utiliser la variance dans les interfaces pour les collections génériques
Variance dans les délégués
Utiliser la variance dans les délégués
Utiliser la variance pour les délégués génériques Func et Action
Arborescences de l’expression
Vue d'ensemble
Comment : exécuter des arborescences d’expression
Comment : modifier des arborescences d'expression
Comment : utiliser des arborescences d’expression pour générer des requêtes dynamiques
Débogage d’arborescences d’expressions dans Visual Studio
Syntaxe de DebugView
Iterators
LINQ (Language Integrated Query)
Vue d'ensemble
Mise en route de LINQ en C#
Introduction aux requêtes LINQ
LINQ et types génériques
Opérations de requête LINQ de base
Transformations de données avec LINQ
Relations des types dans des opérations de requête LINQ
Syntaxe de requête et syntaxe de méthode dans LINQ
Fonctionnalités C# qui prennent en charge LINQ
Procédure pas à pas : Écrire des requêtes en C# (LINQ)
Vue d'ensemble des opérateurs de requête standard
Vue d'ensemble

Syntaxe des expressions de requête pour les opérateurs de requête standard
Classification des opérateurs de requête standard en fonction de leur mode d'exécution
Tri des données
Opérations ensemblistes
Filtrage des données
Opérations de quantificateur
Opérations de projection
Partitionnement des données
Opérations de jointure
Regroupement de données
Opérations de génération
Opérations d'égalité
Opérations d'élément
Conversion des types de données
Opérations de concaténation
Opérations d'agrégation
LINQ to Objects
Vue d'ensemble
LINQ et chaînes
Articles de procédure
Guide pratique pour compter les occurrences d'un mot dans une chaîne (LINQ)
Comment rechercher des phrases qui contiennent un groupe de mots spécifié (LINQ)
Comment rechercher des caractères dans une chaîne (LINQ)
Guide pratique pour combiner des requêtes LINQ avec des expressions régulières
Comment rechercher la différence définie entre deux listes (LINQ)
Comment trier ou filtrer des données texte par mot ou par champ (LINQ)
Comment réorganiser les champs d'un fichier délimité (LINQ)
Guide pratique pour combiner et comparer des collections de chaînes (LINQ)
Comment remplir des collections d'objets issues de plusieurs sources (LINQ)
Comment diviser un fichier en plusieurs fichiers à l'aide de groupes (LINQ)

Comment joindre du contenu de différents fichiers (LINQ)
Guide pratique pour calculer des valeurs de colonnes dans un fichier texte CSV (LINQ)
LINQ et Reflection
Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ)
LINQ et répertoires de fichiers
Vue d'ensemble
Comment rechercher des fichiers avec un attribut ou un nom spécifié
Comment regrouper des fichiers par extension (LINQ)
Comment rechercher le nombre total d'octets dans un ensemble de dossiers (LINQ)
Guide pratique pour comparer le contenu de deux dossiers (LINQ)
Comment rechercher le ou les plus gros fichiers dans une arborescence de répertoires (LINQ)
Comment rechercher les fichiers en double dans une arborescence de répertoires (LINQ)
Guide pratique pour interroger le contenu de fichiers dans un dossier (LINQ)
Comment interroger un ArrayList avec LINQ
Guide pratique pour ajouter des méthodes personnalisées pour les requêtes LINQ
LINQ to ADO.NET (page de portail)
Activation d'une source de données pour l'interrogation LINQ
Outils et IDE Visual Studio prenant en charge LINQ
Réflexion
Sérialisation (C#)
Vue d'ensemble
Comment écrire des données d'objet dans un fichier XML
Comment lire des données d'objet dans un fichier XML
Procédure pas à pas : Persistance d'un objet dans Visual Studio
Instructions, expressions et opérateurs
Vue d'ensemble
Instructions
Membres expression-bodied
Fonctions anonymes

Vue d'ensemble
Comment utiliser des expressions lambda dans une requête
Égalité et comparaisons d'égalité
Comparaisons d'égalité
Comment définir une égalité de valeurs pour un type
Comment tester l'égalité des références (Identité)
Types
Utiliser et définir des types
Cast et conversions de types
Conversion boxing et unboxing
Comment convertir un tableau d'octets en entier
Comment convertir une chaîne en nombre
Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques
Utilisation du type dynamic
Procédure pas à pas : Création et utilisation d'objets dynamiques (C# et Visual Basic)
Classes et structs
Vue d'ensemble
Classes
Objets
Héritage
Polymorphisme
Vue d'ensemble
Versioning avec les mots clés override et new
Savoir quand utiliser les mots clés override et new
Comment : substituer la méthode ToString
Membres
Vue d'ensemble des membres
Classes abstract et sealed et membres de classe
Classes statiques et membres de classe statique
Modificateurs d'accès
Champs
Constantes

Comment : définir des propriétés abstraites
Comment : définir des constantes en C#
Propriétés
Vue d’ensemble des propriétés
Utilisation de propriétés
Propriétés de l’interface
Restriction d'accessibilité de l'accesseur
Comment : déclarer et utiliser des propriétés en lecture-écriture
Propriétés implémentées automatiquement
Guide pratique pour implémenter une classe Lightweight avec des propriétés implémentées automatiquement
Méthodes
Vue d’ensemble des méthodes
Fonctions locales
Retours ref et variables locales ref
Paramètres
Passage de paramètres
Passage de paramètres de type valeur
Passage de paramètres de type référence
Comment : différencier le passage d'un struct et le passage d'une référence de classe à une méthode
Variables locales implicitement typées
Comment : utiliser des tableaux et des variables locales implicitement typés dans une expression de requête
Méthodes d’extension
Comment : implémenter et appeler une méthode d’extension personnalisée
Comment : créer une méthode pour une énumération
Arguments nommés et facultatifs
Comment : utiliser des arguments nommés et facultatifs dans la programmation Office
Constructeurs
Vue d’ensemble des constructeurs
Utilisation de constructeurs

Constructeurs d’instances
Constructeurs privés
Constructeurs statiques
Comment : écrire un constructeur de copie
Finaliseurs
Initialiseurs d’objets et de collections
Comment initialiser des objets à l’aide d’un initialiseur d’objet
Comment initialiser un dictionnaire avec un initialiseur de collection
Types imbriqués
Classes et méthodes partielles
Types anonymes
Comment : retourner des sous-ensembles de propriétés d'éléments dans une requête
Interfaces
Vue d'ensemble
Implémentation d'interface explicite
Comment implémenter explicitement des membres d’interface
Comment implémenter explicitement des membres de deux interfaces
Délégués
Vue d'ensemble
Utilisation de délégués
Délégués avec méthodes nommées et Méthodes anonymes
Comment combiner des délégués (délégués multicast) (Guide de programmation C#)
Comment déclarer, instancier et utiliser un délégué
Tableaux
Vue d'ensemble
Tableaux comme objets
Tableaux unidimensionnels
Tableaux multidimensionnels
Tableaux en escalier
Utiliser foreach avec des tableaux
Passage de tableaux en tant qu’arguments

Tableaux implicitement typés
Chaînes
Programmation avec des chaînes
Comment déterminer si une chaîne représente une valeur numérique
Indexeurs
Vue d'ensemble
Utilisation d'indexeurs
Indexeurs dans les interfaces
Comparaison entre propriétés et indexeurs
Événements
Vue d'ensemble
Comment s’abonner à des événements et s’en désabonner
Comment publier des événements conformes aux indications de .NET
Comment déclencher des événements de la classe de base dans les classes dérivées
Comment implémenter des événements d’interface
Comment implémenter des accesseurs d’événement personnalisés
Génériques
Vue d'ensemble
Paramètres de type générique
Contraintes sur les paramètres de type
Classes génériques
Interfaces génériques
Méthodes génériques
Génériques et tableaux
Délégués génériques
Différences entre les modèles C++ et les génériques C#
Génériques dans le runtime
Génériques et réflexion
Génériques et attributs
Espaces de noms
Vue d'ensemble
Using Namespaces

Comment utiliser l’espace de noms My
Pointeurs et code unsafe
Vue d’ensemble et restrictions
Mémoires tampons de taille fixe
Types de pointeur
Vue d'ensemble
Conversions de pointeurs
Comment utiliser des pointeurs pour copier un tableau d’octets
Commentaires sur la documentation XML
Vue d'ensemble
Balises recommandées pour les commentaires de documentation
Traitement du fichier XML
Délimiteurs pour les balises de documentation
Comment utiliser les fonctionnalités de la documentation XML
Informations de référence sur les balises de la documentation
<c>
<code>
cref, attribut
<example>
<exception>
<include>
<inheritdoc>
<list>
<para>
<param>
<paramref>
<permission>
<remarks>
<returns>
<see>
<seealso>
<summary>

<typeparam>
<typeparamref>
<value>
Exceptions et gestion des exceptions
Vue d'ensemble
Utilisation d'exceptions
Gestion des exceptions
Création et levée d’exceptions
Exceptions générées par le compilateur
Comment gérer une exception à l’aide de try-catch
Comment exécuter le code de nettoyage à l’aide de finally
Comment intercepter une exception non-CLS
Système de fichiers et Registre
Vue d'ensemble
Comment itérer au sein d’une arborescence de répertoires
Comment obtenir des informations sur les fichiers, dossiers et lecteurs
Comment créer un fichier ou un dossier
Comment copier, supprimer et déplacer des fichiers et dossiers
Comment fournir une boîte de dialogue de progression pour les opérations de fichier
Comment écrire dans un fichier texte
Comment lire un fichier texte
Comment lire un fichier texte ligne par ligne
Comment créer une clé dans le Registre
Interopérabilité
Interopérabilité .NET
Vue d'ensemble de l'interopérabilité
Comment accéder aux objets Office Interop à l’aide des fonctionnalités C#
Comment utiliser des propriétés indexées dans la programmation COM Interop
Comment utiliser l’appel de code non managé pour lire un fichier WAV
Procédure pas à pas : Programmation Office (C# et Visual Basic)
Exemple de classe COM
Informations de référence sur le langage

- Vue d'ensemble
- Configurer la version de langage
- Types
  - Types de valeur
    - Vue d'ensemble
    - Types numériques intégraux
    - Types numériques à virgule flottante
    - Conversions numériques intégrées
    - bool
    - char
    - Types d'énumération
    - Types de structure
    - Types de tuples
    - Types valeur Nullable
  - Types référence
    - Caractéristiques des types référence
    - Types référence intégrés
    - class
    - interface
    - Types références Nullables
  - void
  - var
  - Types intégrés
  - Types non managés
  - Valeurs par défaut
- Mots clés
  - Vue d’ensemble
  - Modificateurs
    - Modificateurs d’accès
      - Aide-mémoire
      - Niveaux d'accessibilité
      - Domaine d’accessibilité

Restrictions concernant l’utilisation des niveaux d’accessibilité
internal
private
protected
public
protected internal
protégé privé
abstract
async
const
event
extern
in (modificateur générique)
new (modificateur de membre)
out (modificateur générique)
override
readonly
sealed
static
unsafe
virtual
volatile
Mots clés d'instructions
Catégories d’instructions
Instructions de sélection
if-else
switch
Instructions d'itération
do
for
foreach, in
while

- Instructions de saut
  - break
  - continue
  - goto
  - return
- Instructions de gestion des exceptions
  - throw
  - try-catch
  - try-finally
  - try-catch-finally
- Checked et unchecked
  - Vue d'ensemble
  - checked
  - unchecked
- fixed, instruction
- lock, instruction

Paramètres de méthodes
- Passage de paramètres
- params
- in (modificateur de paramètre)
- ref
- out (modificateur de paramètre)

Mots clés d’espaces de noms
- namespace
- using
  - Contextes pour using
  - using, directive
  - using static, directive
  - using, instruction
- extern alias

Mots clés de test de type
- is

Mots clés de contrainte de type générique
- new, contrainte
- where

Mots clés d'accès
- base
- this

Mots clés littéraux
- null
- true et false
- default

Mots clés contextuels
- Aide-mémoire
- add
- get
- partial, Type
- partial (méthode)
- remove
- set
- when (condition de filtre)
- value
- yield

Mots clés de requête
- Aide-mémoire
- from, clause
- where, clause
- select, clause
- group, clause
- into
- orderby, clause
- join, clause
- let, clause
- ascending

descending
on
equals
by
in
Opérateurs et expressions
Vue d'ensemble
Opérateurs arithmétiques
Opérateurs logiques booléens
Opérateurs de bits et de décalage
Opérateurs d'égalité
Opérateurs de comparaison
Opérateurs et expressions d’accès aux membres
Opérateurs de test de type et expression cast
Opérateurs de conversion définie par l’utilisateur
Opérateurs liés aux pointeurs
Opérateurs d'assignation
Expressions lambda
+ et +=, opérateurs
- et -=, opérateurs
?:, opérateur
! (null-forgiving), opérateur
?? et ??=, opérateurs
=>, opérateur
:: (opérateur)
Opérateur await
Expressions de valeur par défaut
Opérateur delegate
nameof, expression
Opérateur new
Opérateur sizeof
stackalloc, expression

switch, expression
opérateurs true et false
avec une expression
Surcharge d’opérateur
Caractères spéciaux
Vue d'ensemble
$ -- interpolation de chaîne
@ -- identificateur textuel
Attributs lus par le compilateur
Attributs globaux
Général
Informations relatives à l’appelant
Analyse statique Nullable
Directives de préprocesseur
Vue d'ensemble
#if
#else
#elif
#endif
#define
#undef
#warning
#error
#line
#Nullable
#region
#endregion
#pragma
#pragma warning
#pragma checksum
Options du compilateur
Vue d'ensemble

Génération à partir de la ligne de commande avec csc.exe

Comment : définir des variables d'environnement pour la ligne de commande Visual Studio

Options du compilateur C# par catégorie

Options du compilateur C# par ordre alphabétique

@

-addmodule

-appconfig

-baseaddress

-bugreport

-checked

-codepage

-debug

-define

-delaysign

-deterministic

-doc

-errorreport

-filealign

-fullpaths

-help, -?

-highentropyva

-keycontainer

-keyfile

-langversion

-lib

-link

-linkresource

-main

-moduleassemblyname

-noconfig

-nologo

-nostdlib

- -nowarn
- -nowin32manifest
- -nullable
- -optimize
- -out
- -pathmap
- -pdb
- -platform
- -preferreduilang
- -publicsign
- -recurse
- -reference
- -refout
- -refonly
- -resource
- -subsystemversion
- -target
  - -target:appcontainerexe
  - -target:exe
  - -target:library
  - -target:module
  - -target:winexe
  - -target:winmdobj
- -unsafe
- -utf8output
- -warn
- -warnaserror
- -win32icon
- -win32manifest
- -win32res

Erreur du compilateur

Projet de spécification C# 6.0

Propositions C# 7.0 - 9.0

Procédures pas à pas

Introduction au langage C# et à .NET

02/11/2020 • 8 minutes to read • Edit Online

C# est un langage orienté objet élégant et de type sécurisé. C# permet aux développeurs de créer de nombreux types d'applications sécurisées et fiables qui s'exécutent dans l'écosystème .NET.

langage C#

La syntaxe C# est très expressif, mais elle est également simple et facile à apprendre. La syntaxe de l'accolade de C# sera immédiatement reconnaissable à toute personne connaissant le langage C, C++, Java ou JavaScript. Les développeurs qui connaissent l'un de ces langages sont généralement en mesure de travailler de manière productive en C# dans un laps de temps. C# fournit des fonctionnalités puissantes telles que les types Nullable, les délégués, les expressions lambda, les critères spéciaux et l'accès direct à la mémoire sécurisée. C# prend en charge des méthodes et des types génériques, ce qui améliore la sécurité et les performances de type. C# fournit des itérateurs, qui permettent aux implémenteurs de classes de collection de définir des comportements personnalisés pour le code client. Les expressions LINQ (Language-Integrated Query) font de la requête fortement typée une construction de langage de première classe.

En tant que langage orienté objet, C# prend en charge les concepts d'encapsulation, d'héritage et de polymorphisme. Une classe peut hériter directement d'une classe parent, mais peut implémenter un nombre quelconque d'interfaces. Les méthodes qui substituent des méthodes virtuelles dans une classe parente requièrent le mot-clé `override` pour éviter toute redéfinition accidentelle. En C#, un struct est semblable à une classe légère ; Il s'agit d'un type alloué par la pile qui peut implémenter des interfaces mais ne prend pas en charge l'héritage. C# fournit également des enregistrements, qui sont des types de classe dont l'objectif est principalement le stockage des valeurs de données.

C# facilite le développement de composants logiciels à travers plusieurs constructions de langage innovantes, notamment :

- Des signatures de méthode encapsulées appelées *délégués*, qui activent les notifications d'événement de type sécurisé.
- Les propriétés, qui sont utilisées comme accesseurs pour les variables de membre privé.
- Les attributs, qui fournissent des métadonnées déclaratives sur les types au moment de l'exécution.
- Commentaires de documentation XML inline.
- Language-Integrated query (LINQ), qui fournit des fonctionnalités de requête intégrées sur différents types de sources de données.
- Critères spéciaux, qui permettent de contrôler le workflow en inspectant les types de données et les valeurs.

Vous interagissez avec les composants natifs à l'aide d'un processus appelé « Interop ». L'interopérabilité permet aux programmes C# de faire presque tout ce qu'une application native C++ peut faire. C# prend même en charge les pointeurs et le concept de code « unsafe » dans les cas où l'accès direct à la mémoire est essentiel.

Le processus de génération de C# est simple par rapport à C et C++ et plus souple qu'avec Java. Il n'existe aucun fichier d'en-tête distinct, et les types et méthodes n'ont pas à être déclarés dans un ordre particulier. Un fichier source C# peut définir un nombre quelconque de classes, structs, interfaces et événements.

Voici des ressources C# supplémentaires :

- Pour une bonne présentation générale du langage, consultez la visite guidée de C#.
- Pour plus d'informations sur des aspects spécifiques du langage C#, consultez la Référence de C#.
- Pour plus d'informations sur LINQ, consultez LINQ (Language-Integrated Query).

Architecture de la plateforme .NET

Les programmes C# s'exécutent sur .NET, un système d'exécution virtuel appelé le common language runtime (CLR) et un ensemble unifié de bibliothèques de classes. Le CLR est l'implémentation commerciale de Microsoft de l'infrastructure de langage commun (CLI), une norme internationale. L'interface CLI est la base de la création d'environnements d'exécution et de développement dans lesquels les langages et les bibliothèques fonctionnent ensemble de façon transparente.

Le code source écrit en C# est compilé dans un langage intermédiaire (il) conforme à la spécification CLI. Le code de langage intermédiaire et les ressources, telles que les bitmaps et les chaînes, sont stockés dans un assembly, généralement avec une extension. dll. Un assembly contient un manifeste qui fournit des informations sur les types, la version et la culture de l'assembly.

Lorsque le programme C# est exécuté, l'assembly est chargé dans le CLR. Le CLR effectue une compilation juste-à-temps (JIT) pour convertir le code de langage intermédiaire en instructions machine natives. Le CLR fournit d'autres services liés à la garbage collection automatique, à la gestion des exceptions et à la gestion des ressources. Le code exécuté par le CLR est parfois appelé « code managé », contrairement à « code non managé », qui est compilé dans un langage machine natif ciblant un système spécifique.

L'interopérabilité des langages est une fonctionnalité clé de .NET. Le code de langage intermédiaire produit par le compilateur C# est conforme à la spécification de type commun (CTS). Le code de langage intermédiaire généré à partir de C# peut interagir avec le code généré à partir des versions .NET de F #, Visual Basic, C++ ou l'un des plus de 20 autres langages compatibles CTS. Un seul assembly peut contenir plusieurs modules écrits dans différents langages .NET, et les types peuvent se référencer mutuellement comme s'ils avaient été écrits dans le même langage.

Outre les services d'exécution, .NET comprend également des bibliothèques étendues. Ces bibliothèques prennent en charge de nombreuses charges de travail différentes. Elles sont organisées en espaces de noms qui fournissent un large éventail de fonctionnalités utiles pour tout, de l'entrée de fichier à la sortie de la manipulation de chaînes à l'analyse XML, aux infrastructures d'application Web pour Windows Forms contrôles. L'application C# classique utilise largement la bibliothèque de classes .NET pour gérer les tâches courantes de « plomberie ».

Pour plus d'informations sur .NET, consultez vue d'ensemble de .net.

Visite guidée du langage C#

02/11/2020 • 23 minutes to read • Edit Online

C# (prononcé « voir Sharp ») est un langage de programmation moderne, orienté objet et de type sécurisé. C# prend sa source dans la famille de langages C et sera immédiatement reconnaissable aux programmeurs en C, C++, Java et JavaScript. Cette visite guidée fournit une vue d'ensemble des principaux composants du langage dans C# 8 et versions antérieures. Si vous souhaitez explorer le langage par le biais d'exemples interactifs, consultez la présentation des didacticiels C# .

C# est un langage ***de programmation orienté objet et orienté objet*** . C# fournit des constructions de langage pour prendre en charge directement ces concepts, en faisant de C# un langage naturel dans lequel créer et utiliser des composants logiciels. Depuis son origine, C# a ajouté des fonctionnalités pour prendre en charge de nouvelles charges de travail et des pratiques de conception de logiciels émergentes.

Plusieurs fonctionnalités C# vous aident à construire des applications robustes et durables. Le ***garbage collection*** libère automatiquement la mémoire occupée par des objets inutilisés inaccessibles. La ***gestion des exceptions*** offre une approche structurée et extensible de la détection et de la récupération des erreurs. Les ***expressions lambda*** prennent en charge les techniques de programmation fonctionnelle. La ***syntaxe de requête*** crée un modèle commun pour travailler avec des données de n'importe quelle source. La prise en charge linguistique pour les ***opérations asynchrones*** fournit la syntaxe pour la création de systèmes distribués. Les ***critères spéciaux*** fournissent une syntaxe permettant de séparer facilement les données des algorithmes dans les systèmes distribués modernes. C# possède un ***système de type unifié***. Tous les types C#, y compris les types primitifs tels que `int` et `double`, héritent d'un seul type `object` racine. Tous les types partagent un ensemble d'opérations courantes. Les valeurs de n'importe quel type peuvent être stockées, transportées et exploitées de manière cohérente. En outre, C# prend en charge les types référence définis par l'utilisateur et les types valeur. C# permet l'allocation dynamique d'objets et le stockage en ligne de structures légères.

C# met l'accent sur le contrôle de ***version*** pour s'assurer que les programmes et les bibliothèques peuvent évoluer au fil du temps d'une manière compatible. Les aspects de la conception de C# qui ont été directement influencés par les considérations relatives à la gestion des versions incluent les `virtual` `override` modificateurs et séparés, les règles de résolution de surcharge de méthode et la prise en charge des déclarations de membres d'interface explicites.

Hello World

Le programme « Hello, World » est souvent utilisé pour présenter un langage de programmation. Le voici en C# :

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Le programme « Hello, World » commence par une directive `using` qui fait référence à l'espace de noms `System`. Les espaces de noms représentent un moyen hiérarchique d'organiser les bibliothèques et les programmes C#. Les espaces de noms contiennent des types et d'autres espaces de noms ; par exemple, l'espace de noms `System` contient plusieurs types, notamment la classe `Console` référencée dans le programme, et d'autres espaces de

noms, tels que `IO` et `Collections`. Une directive `using` qui fait référence à un espace de noms donné permet l'utilisation non qualifiée des types membres de cet espace de noms. En raison de la directive `using`, le programme peut utiliser `Console.WriteLine` comme raccourci pour `System.Console.WriteLine`.

La classe `Hello` déclarée par le programme « Hello, World » a un membre unique, la méthode nommée `Main`. La `Main` méthode est déclarée avec le `static` modificateur. Si les méthodes d'instance peuvent faire référence à une instance d'objet englobante particulière avec le mot clé `this`, les méthodes statiques fonctionnent sans référence à un objet particulier. Par Convention, une méthode statique nommée `Main` sert de point d'entrée d'un programme C#.

La sortie du programme est générée par la méthode `WriteLine` de la classe `Console` dans l'espace de noms `System`. Cette classe est fournie par les bibliothèques de classes standard, qui, par défaut, sont référencées automatiquement par le compilateur.

Types et variables

Il existe deux genres de types en C# : les *types référence* et les *types valeur*. Les variables des types valeur contiennent directement leurs données alors que les variables des types référence contiennent des références à leurs données, connues sous le nom d'objets. Avec les types référence, il est possible que deux variables référencent le même objet et que les opérations sur une variable affectent l'objet référencé par l'autre variable. Avec les types valeur, les variables ont chacune leur propre copie des données, et il n'est pas possible pour les opérations sur l'une d'affecter l'autre (à l'exception des `ref` `out` variables de paramètre et).

Un ***identificateur*** est un nom de variable. Un identificateur est une séquence de caractères Unicode sans espace blanc. Un identificateur peut être un mot réservé C#, s'il est préfixé par `@` . Cela peut être utile lors de l'interaction avec d'autres langages.

Les types valeur de C# sont divisés en *types simples*, *types ENUM*, *types struct*, types *valeur Nullable* et *types valeur de tuple*. Les types référence de C# sont encore divisés en *types classe*, *types interface*, *types tableau*et *types délégués*.

Le schéma suivant fournit une vue d'ensemble du système de type de C#.

- Types de valeur
  - Types simples
    - Intégrale signé: `sbyte` , `short` , `int` , `long`
    - Entier non signé: `byte` , `ushort` , `uint` , `ulong`
    - Caractères Unicode: `char` , qui représente une unité de code UTF-16
    - Virgule flottante binaire IEEE: `float` , `double`
    - Virgule flottante décimale haute précision: `decimal`
    - Boolean : `bool` , qui représente des valeurs booléennes, les valeurs qui sont `true` ou `false`
  - Types ENUM
    - Types définis par l'utilisateur sous la forme `enum E {...}` . Un type `enum` est un type distinct avec des constantes nommées. Chaque type `enum` a un type sous-jacent qui doit être un des huit types intégraux. L'ensemble de valeurs d'un type `enum` est le même que l'ensemble de valeurs du type sous-jacent.
  - Types struct
    - Types définis par l'utilisateur de la forme `struct S {...}`
  - Types valeur Nullable
    - Extensions de tous les autres types de valeurs avec une valeur `null`
  - Types de valeur de Tuple
    - Types définis par l'utilisateur de la forme `(T1, T2, ...)`

- Types référence
  - Types de classe
    - Classe de base fondamentale de tous les autres types : `object`
    - Chaînes Unicode: `string` , qui représente une séquence d'unités de code UTF-16
    - Types définis par l'utilisateur de la forme `class C {...}`
  - Types d'interface
    - Types définis par l'utilisateur de la forme `interface I {...}`
  - Types tableau
    - Unidimensionnel, unidimensionnel et en escalier. Par exemple : `int[]` , `int[,]` et `int[][]`
  - Types délégués
    - Types définis par l'utilisateur de la forme `delegate int D(...)`

Les programmes C# utilisent les *déclarations de type* pour créer de nouveaux types. Une déclaration de type spécifie le nom et les membres du nouveau type. Six des catégories de types C# sont définissables par l'utilisateur : types de classes, types struct, types interface, types ENUM, types délégués et types valeur de Tuple.

- Un type `class` définit une structure de données qui contient des données membres (champs) et des fonctions membres (méthodes, propriétés, etc.). Les types de classes prennent en charge l'héritage unique et le polymorphisme, des mécanismes par lesquels les classes dérivées peuvent étendre et spécialiser les classes de base.
- Un type `struct` est similaire à un type de classe dans la mesure où il représente une structure avec des membres de données et des membres de fonctions. Toutefois, contrairement aux classes, les structs sont des types valeur et ne nécessitent généralement pas d'allocation de tas. Les types struct ne prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object` .
- Un `interface` type définit un contrat comme un jeu nommé de membres publics. Un `class` ou `struct` qui implémente un `interface` doit fournir des implémentations des membres de l'interface. Une `interface` peut hériter de plusieurs interfaces de base, et une `class` ou `struct` peut implémenter plusieurs interfaces.
- Un type `delegate` représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont semblables aux types de fonction fournis par les langages fonctionnels. Elles sont également similaires au concept de pointeurs de fonction qui se trouvent dans d'autres langages. Contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

Les `class` types,, `struct` `interface` et `delegate` prennent tous en charge les génériques, dans lesquels ils peuvent être paramétrés avec d'autres types.

C# prend en charge les tableaux unidimensionnels et multidimensionnels de tout type. Contrairement aux types indiqués ci-dessus, les types de tableau ne doivent pas nécessairement être déclarés avant de pouvoir être utilisés. Au lieu de cela, les types de tableaux sont construits en ajoutant des crochets à un nom de type. Par exemple, `int[]` est un tableau unidimensionnel de `int` , `int[,]` est un tableau à deux dimensions de `int` , et `int[][]` est un tableau unidimensionnel de tableaux unidimensionnels, ou un tableau en escalier, de `int` .

Les types Nullable ne nécessitent pas une définition distincte. Pour chaque type non Nullable `T` , il existe un type Nullable correspondant `T?` qui peut contenir une valeur supplémentaire, `null` . Par exemple, `int?` est un type qui peut contenir n'importe quel entier 32 bits ou la valeur `null` , et `string?` est un type qui peut contenir n'importe quelle `string` valeur ou `null` .

Le système de type C# est unifié de telle façon qu'une valeur de n'importe quel type peut être traitée comme un `object` . Chaque type dans C# dérive directement ou indirectement du type `object` , et `object` est la classe de base fondamentale de tous les types. Les valeurs des types référence sont considérées comme des objets simplement en affichant les valeurs en tant que type `object` . Les valeurs des types valeur sont considérées comme des objets en effectuant des opérations de *boxing* et *d'unboxing*. Dans l'exemple suivant, une valeur `int`

est convertie en `object` et à nouveau en `int`.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

Quand une valeur d'un type de valeur est assignée à une `object` référence, un « Box » est alloué pour contenir la valeur. Cette zone est une instance d'un type référence, et la valeur est copiée dans cette zone. À l'inverse, lorsqu'une `object` référence est castée en un type valeur, une vérification est effectuée que le référencé `object` est une zone du type valeur correct. Si la vérification est réussie, la valeur dans la zone est copiée dans le type valeur.

Le système de type unifié de C# signifie que les types valeur sont traités comme `object` des références « à la demande ». En raison de l'unification, les bibliothèques à usage général qui utilisent le type `object` peuvent être utilisées avec tous les types qui dérivent de `object` , y compris les types référence et les types valeur.

Il existe plusieurs types de *variables* en C#, y compris les champs, les éléments de tableau, les variables locales et les paramètres. Les variables représentent des emplacements de stockage. Chaque variable a un type qui détermine les valeurs qui peuvent être stockées dans la variable, comme indiqué ci-dessous.

- Type de valeur n'acceptant pas Null
  - Une valeur de ce type exact
- Types valeur Nullable
  - Une valeur `null` ou une valeur de ce type exact
- object
  - Une référence `null`, une référence à un objet de tout type référence ou une référence à une valeur boxed de n'importe quel type valeur
- Type de classe
  - Une référence `null`, une référence à une instance de ce type de classe ou une référence à une instance d'une classe dérivée de ce type de classe
- Type d'interface
  - Une référence `null`, une référence à une instance d'un type de classe qui implémente ce type d'interface ou une référence à une valeur boxed d'un type valeur qui implémente ce type d'interface
- Type tableau
  - Une référence `null`, une référence à une instance de ce type de tableau ou une instance d'un type de tableau compatible
- Type délégué
  - Une référence `null` ou une référence à une instance d'un type délégué compatible

Structure du programme

Les concepts clés de l'organisation en C# sont les ***programmes***, les ***espaces de noms***, les ***types***, ***les membres***et les ***assemblys***. Les programmes déclarent des types qui contiennent des membres et peuvent être organisés en espaces de noms. Les classes, les structs et les interfaces sont des exemples de types. Les champs, méthodes, propriétés et événements sont des exemples de membres. Lorsque les programmes C# sont compilés, ils sont empaquetés physiquement dans des assemblys. Les assemblys ont généralement l'extension de fichier `.exe` ou `.dll`, selon qu'elles implémentent des ***applications*** ou des ***bibliothèques***, respectivement.

Prenons l'exemple d'un assembly qui contient le code suivant :

```
using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}
```

Le nom qualifié complet de cette classe est `Acme.Collections.Stack`. La classe contient plusieurs membres : un champ nommé `top`, deux méthodes nommées `Push` et `Pop`, et une classe imbriquée nommée `Entry`. La classe `Entry` contient trois membres en plus : un champ nommé `next`, un autre nommé `data` et un constructeur. `Stack` Est une classe *générique* . Il a un paramètre de type, `T` qui est remplacé par un type concret lorsqu'il est utilisé.

> **NOTE**
> Une *pile* est une collection « First in-Last out » (Filo). De nouveaux éléments sont ajoutés en haut de la pile. Lorsqu'un élément est supprimé, il est supprimé du haut de la pile.

Les assemblys contiennent le code exécutable sous forme d'instructions de langage intermédiaire (IL) et des informations symboliques sous la forme de métadonnées. Avant son exécution, le compilateur juste-à-temps (JIT) du Common Language Runtime .NET convertit le code IL dans un assembly en code spécifique au processeur.

Étant donné qu'un assembly est une unité autodescriptive de fonctionnalité contenant à la fois du code et des métadonnées, il n'est pas nécessaire d'utiliser des `#include` directives et des fichiers d'en-tête en C#. Les membres et types publics contenus dans un assembly particulier sont disponibles dans un programme C# par simple référence à cet assembly lors de la compilation du programme. Par exemple, ce programme utilise la classe `Acme.Collections.Stack` à partir de l'assembly `acme.dll` :

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

Pour compiler ce programme, vous devez *référencer* l'assembly contenant la classe Stack définie dans l'exemple précédent.

Les programmes C# peuvent être stockés dans plusieurs fichiers sources. Lorsqu'un programme C# est compilé, tous les fichiers sources sont traités ensemble, et les fichiers sources peuvent se référencer librement. Conceptuellement, c'est comme si tous les fichiers sources étaient concaténés dans un fichier volumineux avant d'être traités. Les déclarations anticipées ne sont jamais nécessaires en C#, car, à quelques exceptions près, l'ordre de déclaration n'est pas significatif. C# ne limite pas un fichier source à la déclaration d'un seul type public et n'exige pas que le nom du fichier source corresponde à un type déclaré dans le fichier source.

D'autres Articles de cette présentation expliquent ces blocs organisationnels.

NEXT

Types et membres

16/12/2020 • 11 minutes to read • Edit Online

Classes et objets

Les *Classes* sont le type le plus fondamental de C#. Une classe est une structure de données qui combine l'état (champs) et les actions (méthodes et autres fonctions membres) en une seule unité. Une classe fournit une définition pour les *instances* de la classe, également appelées *objets*. Les classes prennent en charge *l'héritage* et le *polymorphisme*, des mécanismes par lesquels les *classes dérivées* peuvent étendre et spécialiser les *classes de base*.

Les nouvelles classes sont créées à l'aide des déclarations de classe. Une déclaration de classe commence par un en-tête. L'en-tête spécifie :

- Attributs et modificateurs de la classe
- Nom de la classe.
- Classe de base (en cas d'héritage d'une classe de base)
- Interfaces implémentées par la classe.

L'en-tête est suivi par le corps de la classe, qui se compose d'une liste de déclarations de membres écrites entre les délimiteurs `{` et `}`.

Le code suivant illustre une déclaration d'une classe simple nommée `Point` :

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Les instances de classes sont créées à l'aide de l'opérateur `new`, qui alloue la mémoire pour une nouvelle instance, appelle un constructeur pour initialiser l'instance et retourne une référence à l'instance. Les instructions suivantes créent deux `Point` objets et stockent les références à ces objets dans deux variables :

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

La mémoire occupée par un objet est automatiquement libérée lorsque l'objet n'est plus accessible. Il n'est ni nécessaire ni possible de libérer explicitement des objets en C#.

Paramètres de type

Les classes génériques définissent les * **paramètres de type** _. Les paramètres de type sont une liste de noms de paramètre de type entre crochets pointus. Les paramètres de type suivent le nom de la classe. Les paramètres de type peuvent ensuite être utilisés dans le corps des déclarations de classe pour définir les membres de la classe. Dans l'exemple suivant, les paramètres de type de `Pair` sont `TFirst` et `TSecond` :

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

Un type de classe déclaré pour prendre des paramètres de type est appelé un _generic type de classe *. Les types struct, interface et Delegate peuvent également être génériques. Lorsque la classe générique est utilisée, des arguments de type doivent être fournis pour chacun des paramètres de type :

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;     // TFirst int
string s = pair.Second; // TSecond string
```

Un type générique avec des arguments de type fournis, comme `Pair<int,string>` ci-dessus, est appelé un *type construit*.

Classes de base

Une déclaration de classe peut spécifier une classe de base. Suivez les paramètres Name et type de la classe avec le signe deux-points et le nom de la classe de base. L'omission d'une spécification de classe de base revient à dériver du type `object`. Dans l'exemple suivant, la classe de base de `Point3D` est `Point` . Dans le premier exemple, la classe de base de `Point` est `object` :

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

Une classe hérite des membres de sa classe de base. L'héritage signifie qu'une classe contient implicitement tous les membres de sa classe de base. Une classe n'hérite pas de l'instance et des constructeurs statiques, et du finaliseur. Une classe dérivée peut ajouter de nouveaux membres aux membres qu'elle hérite, mais elle ne peut pas supprimer la définition d'un membre hérité. Dans l'exemple précédent, `Point3D` hérite des `X` `Y` membres et de `Point` , et chaque `Point3D` instance contient trois propriétés, `X` , `Y` et `Z` .

Il existe une conversion implicite d'un type de classe vers un de ses types de classe de base. Une variable d'un type de classe peut référencer une instance de cette classe ou une instance de toute classe dérivée. Par exemple, pour les déclarations de classe précédentes, une variable de type `Point` peut faire référence à un objet `Point` ou `Point3D` :

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Structures

Les classes définissent des types qui prennent en charge l'héritage et le polymorphisme. Elles vous permettent de créer des comportements sophistiqués basés sur des hiérarchies de classes dérivées. En revanche, les types *

struct _ sont des types plus simples dont l'objectif principal est de stocker des valeurs de données. Les structs ne peuvent pas déclarer un type de base ; ils dérivent implicitement de System.ValueType . Vous ne pouvez pas dériver `struct` d'autres types à partir d'un `struct` type. Ils sont implicitement sealed.

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

Interfaces

Une *interface* définit un contrat qui peut être implémenté par des classes et des structs. Une interface peut contenir des méthodes, des propriétés, des événements et des indexeurs. En général, une interface ne fournit pas d'implémentations des membres qu'elle définit. elle spécifie simplement les membres qui doivent être fournis par les classes ou les structs qui implémentent l'interface.

Les interfaces peuvent utiliser *l'héritage multiple*. Dans l'exemple suivant, l'interface `IComboBox` hérite à la fois de `ITextBox` et `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Les classes et les structs peuvent implémenter plusieurs interfaces. Dans l'exemple suivant, la classe `EditBox` implémente à la fois `IControl` et `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Lorsqu'une classe ou un struct implémente une interface spécifique, les instances de cette classe ou struct peuvent être converties implicitement en ce type d'interface. Par exemple

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Énumérations

Un type *enum* définit un ensemble de valeurs constantes. L'exemple suivant `enum` déclare des constantes qui définissent des légumes racines différents :

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

Vous pouvez également définir un `enum` à utiliser en combinaison comme indicateurs. La déclaration suivante déclare un jeu d'indicateurs pour les quatre saisons. Toute combinaison de saisons peut être appliquée, y compris une `All` valeur qui inclut toutes les saisons :

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

L'exemple suivant montre des déclarations des deux énumérations précédentes :

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Types Nullable

Les variables de tout type peuvent être déclarées comme n' *acceptant pas les valeurs NULL* ou *null*. Une variable Nullable peut contenir une `null` valeur supplémentaire, ce qui n'indique aucune valeur. Les types valeur Nullable (structs ou enums) sont représentés par System.Nullable<T> . Les types référence non Nullable et Nullable sont représentés par le type référence sous-jacent. La distinction est représentée par les métadonnées lues par le compilateur et certaines bibliothèques. Le compilateur fournit des avertissements quand des références Nullable sont déréférencées sans avoir d'abord vérifié leur valeur par rapport à `null` . Le compilateur fournit également des avertissements lorsque des références non Nullable se voient affecter une valeur qui peut être `null` . L'exemple suivant déclare un *entier Nullable*, en l'initialisant à `null` . Ensuite, il définit la valeur sur `5` . Il illustre le même concept avec une *chaîne Nullable*. Pour plus d'informations, consultez types valeur Nullable et types référence Nullable.

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.";
```

Tuples

C# prend en charge les _ *tuples* *, qui fournit une syntaxe concise pour regrouper plusieurs éléments de données dans une structure de données légère. Vous instanciez un tuple en déclarant les types et les noms des membres entre `(` et `)` , comme indiqué dans l'exemple suivant :

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Les tuples offrent une alternative à la structure de données avec plusieurs membres, sans utiliser les blocs de construction décrits dans l'article suivant.

PRÉCÉDENT SUIVANT

Blocs de construction de programme

16/12/2020 • 42 minutes to read • Edit Online

Les types décrits dans l'article précédent sont générés à l'aide des blocs de construction suivants : * **members** _, *expressions* et *instructions*.

Membres

Les membres d'un `class` sont soit des membres *statiques* , soit des *membres d'instance*. Les membres statiques appartiennent à des classes, et les membres d'instance appartiennent à des objets (instances de classes).

La liste suivante fournit une vue d'ensemble des types de membres qu'une classe peut contenir.

- _ * Constantes * * : valeurs constantes associées à la classe
- **Fields**: variables associées à la classe
- **Méthodes**: actions qui peuvent être effectuées par la classe
- **Propriétés**: actions associées à la lecture et à l'écriture des propriétés nommées de la classe
- **Indexeurs**: actions associées à l'indexation d'instances de la classe comme un tableau
- **Événements**: notifications qui peuvent être générées par la classe
- **Opérateurs**: conversions et opérateurs d'expression pris en charge par la classe
- **Constructeurs**: actions requises pour initialiser des instances de la classe ou la classe elle-même
- **Finaliseurs**: actions effectuées avant que les instances de la classe soient définitivement ignorées
- **Types**: types imbriqués déclarés par la classe

Accessibilité

Chaque membre d'une classe a une accessibilité associée, qui contrôle les zones de texte de programme qui peuvent accéder au membre. Il existe six formes possibles d'accessibilité. Les modificateurs d'accès sont résumés ci-dessous.

- `public` : L'accès n'est pas limité.
- `private` : L'accès est limité à cette classe.
- `protected` : L'accès est limité à cette classe ou aux classes dérivées de cette classe.
- `internal` : L'accès est limité à l'assembly actuel (`.exe` ou `.dll`).
- `protected internal` : L'accès est limité à cette classe, aux classes dérivées de cette classe ou aux classes au sein du même assembly.
- `private protected` : L'accès est limité à cette classe ou aux classes dérivées de ce type dans le même assembly.

Champs

Un *champ* est une variable qui est associée à une classe ou une instance d'une classe.

Un champ déclaré avec le modificateur static définit un champ statique. Un champ statique identifie exactement un seul emplacement de stockage. Quel que soit le nombre d'instances d'une classe qui sont créées, il n'y a qu'une seule copie d'un champ statique.

Un champ déclaré sans le modificateur static définit un champ d'instance. Chaque instance d'une classe contient une copie distincte de tous les champs d'instance de cette classe.

Dans l'exemple suivant, chaque instance de la `Color` classe a une copie distincte des `R` champs d' `G` instance, et

`B` , mais il n'existe qu'une seule copie des `Black` `White` `Red` `Green` `Blue` champs statiques,,, et :

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}
```

Comme indiqué dans l'exemple précédent, les *champs en lecture seule* peuvent être déclarés avec un modificateur `readonly` . L'assignation à un champ en lecture seule ne peut se produire que dans le cadre de la déclaration du champ ou dans un constructeur de la même classe.

Méthodes

Une *méthode* est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les *méthodes statiques* sont accessibles à travers la classe. Les *méthodes d'instance* sont accessibles via des instances de la classe.

Les méthodes peuvent avoir une liste de *paramètres*, qui représentent des valeurs ou des références variables passées à la méthode. Les méthodes ont un *type de retour*, qui spécifie le type de la valeur calculée et retournée par la méthode. Le type de retour d'une méthode est `void` s'il ne retourne pas de valeur.

Comme les types, les méthodes peuvent également être un jeu de paramètres de type pour lesquels les arguments de type doivent être spécifiés lorsque la méthode est appelée. Contrairement aux types, les arguments de type peuvent souvent être déduits à partir des arguments d'un appel de méthode et n'ont pas à être fournis explicitement.

La *signature* d'une méthode doit être unique dans la classe dans laquelle la méthode est déclarée. La signature d'une méthode se compose du nom de la méthode, du nombre de paramètres de type, du nombre, des modificateurs et des types de ses paramètres. La signature d'une méthode n'inclut pas le type de retour.

Quand un corps de méthode est une expression unique, la méthode peut être définie à l'aide d'un format d'expression compact, comme indiqué dans l'exemple suivant :

```
public override string ToString() => "This is an object";
```

Paramètres

Les paramètres sont utilisés pour passer des valeurs ou des références variables aux méthodes. Les paramètres d'une méthode obtiennent leurs valeurs réelles à partir des *arguments* qui sont spécifiés lorsque la méthode est appelée. Il existe quatre types de paramètres : les paramètres de valeur, les paramètres de référence, les paramètres de sortie et les tableaux de paramètres.

Un *paramètre de valeur* est utilisé pour passer des arguments en entrée. Un paramètre de valeur correspond à une

variable locale qui obtient sa valeur initiale de l'argument qui a été transmis pour le paramètre. Les modifications apportées à un paramètre de valeur n'affectent pas l'argument qui a été transmis pour le paramètre.

Les paramètres de valeur peuvent être facultatifs, en spécifiant une valeur par défaut afin que les arguments correspondants puissent être omis.

Un *paramètre de référence* est utilisé pour passer des arguments par référence. L'argument passé pour un paramètre de référence doit être une variable avec une valeur définie. Pendant l'exécution de la méthode, le paramètre de référence représente le même emplacement de stockage que la variable d'argument. Un paramètre de référence est déclaré avec le modificateur `ref`. L'exemple suivant illustre l'utilisation des paramètres `ref`.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}");    // "2 1"
}
```

Un *paramètre de sortie* est utilisé pour passer des arguments par référence. Il est similaire à un paramètre de référence, sauf qu'il ne nécessite pas d'affecter explicitement une valeur à l'argument fourni par l'appelant. Un paramètre de sortie est déclaré avec le modificateur `out`. L'exemple suivant montre l'utilisation de paramètres `out` à l'aide de la syntaxe introduite dans C# 7.

```
static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}
```

Un *tableau de paramètres* autorise le passage d'un nombre variable d'arguments à une méthode. Un tableau de paramètres est déclaré avec le modificateur `params`. Seul le dernier paramètre d'une méthode peut être un tableau de paramètres, et le type d'un tableau de paramètres doit être un type tableau unidimensionnel. Les `Write` `WriteLine` méthodes et de la System.Console classe sont de bons exemples d'utilisation des tableaux de paramètres. Elles sont déclarées comme suit.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Dans une méthode qui utilise un tableau de paramètres, le tableau de paramètres se comporte exactement comme un paramètre ordinaire de type tableau. Toutefois, dans un appel d'une méthode avec un tableau de paramètres, il est possible de passer un argument unique du type de tableau de paramètres ou un nombre quelconque

d'arguments du type d'élément du tableau de paramètres. Dans ce cas, une instance de tableau est automatiquement créée et initialisée avec les arguments donnés. Cet exemple

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

revient à écrire ce qui suit.

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Corps de la méthode et variables locales

Le corps d'une méthode spécifie les instructions à exécuter lorsque la méthode est appelée.

Un corps de méthode peut déclarer des variables qui sont spécifiques à l'appel de la méthode. Ces variables sont appelées *variables locales*. Une déclaration de variable locale spécifie un nom de type, un nom de variable et éventuellement une valeur initiale. L'exemple suivant déclare une variable locale `i` avec une valeur initiale de zéro et une variable locale `j` sans valeur initiale.

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

C# requiert qu'une variable locale soit *assignée de manière définitive* avant de pouvoir obtenir sa valeur. Par exemple, si la déclaration de la précédente `i` n'inclut pas de valeur initiale, le compilateur signale une erreur pour les utilisations ultérieures de, `i` car n' `i` est pas définitivement assignée à ces points dans le programme.

Une méthode peut utiliser les instructions `return` pour retourner le contrôle à son appelant. Dans une méthode retournant `void` , les `return` instructions ne peuvent pas spécifier une expression. Dans une méthode avec un type de retour non void, les instructions `return` doivent inclure une expression qui calcule la valeur de retour.

Méthodes statiques et d'instance

Une méthode déclarée avec un `static` modificateur est une *méthode statique*. Une méthode statique ne fonctionne pas sur une instance spécifique et peut uniquement accéder directement aux membres statiques.

Une méthode déclarée sans `static` modificateur est une *méthode d'instance*. Une méthode d'instance opère sur une instance spécifique et peut accéder aux membres statiques et d'instance. L'instance sur laquelle une méthode

d'instance a été appelée est explicitement accessible en tant que `this`. Il s'agit d'une erreur qui consiste à faire référence à `this` dans une méthode statique.

La classe `Entity` suivante a à la fois des statiques et des membres d'instance.

```
class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}
```

Chaque `Entity` instance contient un numéro de série (et probablement d'autres informations qui ne sont pas indiquées ici). Le constructeur `Entity` (qui est similaire à une méthode d'instance) initialise la nouvelle instance avec le numéro de série suivant. Étant donné que le constructeur est un membre d'instance, il est autorisé à accéder à la fois au `_serialNo` champ d'instance et au `s_nextSerialNo` champ statique.

Les méthodes statiques `GetNextSerialNo` et `SetNextSerialNo` peuvent accéder au champ statique `s_nextSerialNo`, mais ce serait une erreur pour eux d'accéder directement au champ d'instance `_serialNo`.

L'exemple suivant illustre l'utilisation de la `Entity` classe.

```
Entity.SetNextSerialNo(1000);
Entity e1 = new Entity();
Entity e2 = new Entity();
Console.WriteLine(e1.GetSerialNo());          // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());          // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());  // Outputs "1002"
```

Les `SetNextSerialNo` `GetNextSerialNo` méthodes statiques et sont appelées sur la classe, tandis que la `GetSerialNo` méthode d'instance est appelée sur des instances de la classe.

Méthodes virtuelles, de substitution et abstraites

Lorsqu'une déclaration de méthode d'instance inclut un modificateur `virtual`, la méthode est appelée *méthode virtuelle*. Si aucun modificateur virtuel n'est présent, la méthode est appelée *méthode non virtuelle*.

Lorsqu'une méthode virtuelle est appelée, le *type au moment de l'exécution* de l'instance pour laquelle cet appel prend place détermine l'implémentation de méthode à appeler. Dans un appel de méthode non virtuelle, le *type lors de la compilation* de l'instance est le facteur déterminant.

Une méthode virtuelle peut être *substituée* dans une classe dérivée. Lorsqu'une déclaration de méthode d'instance

comprend un modificateur override, la méthode substitue une méthode virtuelle héritée ayant la même signature. Une déclaration de méthode virtuelle introduit une nouvelle méthode. Une déclaration de méthode override spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

Une *méthode abstraite* est une méthode virtuelle sans implémentation. Une méthode abstraite est déclarée avec le `abstract` modificateur et est autorisée uniquement dans une classe abstraite. Une méthode abstraite doit être remplacée dans chaque classe dérivée non abstraite.

L'exemple suivant déclare une classe abstraite, `Expression`, qui représente un nœud d'arborescence de l'expression et trois classes dérivées, `Constant`, `VariableReference` et `Operation`, qui implémentent des nœuds d'arborescence de l'expression pour les références variables, les constantes et les opérations arithmétiques. (Cet exemple est similaire à, mais n'est pas associé aux types d'arborescence d'expression).

```
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}
```

Les quatre classes précédentes permet de modéliser des expressions arithmétiques. Par exemple, en utilisant des

instances de ces classes, l'expression `x + 3` peut être représentée comme suit.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

La méthode `Evaluate` d'une instance `Expression` est appelée pour évaluer l'expression donnée et produire une valeur `double`. La méthode prend un argument `Dictionary` qui contient des noms de variables (comme clés des entrées) et des valeurs (comme valeurs des entrées). Comme `Evaluate` est une méthode abstraite, les classes non abstraites dérivées de `Expression` doivent remplacer `Evaluate`.

Une implémentation de `Constant` de `Evaluate` renvoie simplement la constante stockée. Une implémentation de `VariableReference` recherche le nom de variable dans le dictionnaire et renvoie la valeur résultante. Une implémentation de `Operation` évalue d'abord les opérandes de gauche et de droite (en appelant de manière récursive leurs méthodes `Evaluate`), puis effectue l'opération arithmétique donnée.

Le programme suivant utilise les classes `Expression` pour évaluer l'expression `x * (y + 2)` pour différentes valeurs de `x` et `y`.

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new Dictionary<string, object>();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

Surcharge de méthode

La *surcharge* de méthode permet d'avoir plusieurs méthodes dans la même classe avec le même nom, tant qu'elles ont des signatures uniques. Lors de la compilation d'un appel à une méthode surchargée, le compilateur utilise *la résolution de surcharge* pour déterminer la méthode spécifique à appeler. La résolution de surcharge recherche l'unique méthode qui correspond le mieux aux arguments. Si aucune meilleure correspondance ne peut être trouvée, une erreur est signalée. L'exemple suivant montre la résolution de surcharge en action. Le commentaire pour chaque appel de la `UsageExample` méthode indique la méthode appelée.

```
class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine("F<T>(T)");
    static void F(double x, double y) => Console.WriteLine("F(double, double)");

    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Comme indiqué dans l'exemple, une méthode particulière peut toujours être sélectionnée en effectuant un cast explicite des arguments vers les types de paramètres et les arguments de type exacts.

Autres fonctions membres

Les membres qui contiennent du code exécutable sont collectivement regroupés sous les *membres de fonction* d'une classe. La section précédente décrit les méthodes, qui sont les principaux types de fonctions membres. Cette section décrit les autres types de fonctions membres pris en charge par C# : constructeurs, propriétés, indexeurs, événements, opérateurs et finaliseurs.

L'exemple suivant montre une classe générique appelée `MyList<T>` , qui implémente une liste d'objets pouvant être augmentée. La classe contient plusieurs exemples des types les plus courants de membres de fonction.

```
public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get =>  _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
    }
```

```
    public T this[int index]
    {
        get => _items[index];
        set
        {
            _items[index] = value;
            OnChanged();
        }
    }

    public void Add(T item)
    {
        if (_count == Capacity) Capacity = _count * 2;
        _items[_count] = item;
        _count++;
        OnChanged();
    }
    protected virtual void OnChanged() =>
        Changed?.Invoke(this, EventArgs.Empty);

    public override bool Equals(object other) =>
        Equals(this, other as MyList<T>);

    static bool Equals(MyList<T> a, MyList<T> b)
    {
        if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
        if (Object.ReferenceEquals(b, null) || a._count != b._count)
            return false;
        for (int i = 0; i < a._count; i++)
        {
            if (!object.Equals(a._items[i], b._items[i]))
            {
                return false;
            }
        }
        return true;
    }

    public event EventHandler Changed;

    public static bool operator ==(MyList<T> a, MyList<T> b) =>
        Equals(a, b);

    public static bool operator !=(MyList<T> a, MyList<T> b) =>
        !Equals(a, b);
}
```

Constructeurs

C# prend en charge les constructeurs statiques et d'instance. Un *constructeur d'instance* est un membre qui implémente les actions requises pour initialiser une instance d'une classe. Un *constructeur statique* est un membre qui implémente les actions requises pour initialiser une classe lorsqu'elle est chargée pour la première fois.

Un constructeur est déclaré comme une méthode sans aucun type de retour et le même nom que la classe contenante. Si une déclaration de constructeur comprend un `static` modificateur, elle déclare un constructeur statique. Dans le cas contraire, elle déclare un constructeur d'instance.

Les constructeurs d'instance peuvent être surchargés et avoir des paramètres facultatifs. Par exemple, la classe `MyList<T>` déclare un constructeur d'instance avec un seul paramètre `int` facultatif. Les constructeurs d'instance sont appelés en utilisant l'opérateur `new`. Les instructions suivantes allouent deux instances `MyList<string>` à l'aide du constructeur de la classe `MyList` avec et sans l'argument facultatif.

```
MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);
```

Contrairement aux autres membres, les constructeurs d'instance ne sont pas hérités. Une classe n'a pas de constructeurs d'instance autres que ceux qui sont réellement déclarés dans la classe. Si aucun constructeur d'instance n'est fourni pour une classe, un constructeur vide sans paramètre est fourni automatiquement.

Propriétés

Les *propriétés* sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés, et la syntaxe pour accéder aux champs et propriétés est la même. Toutefois, contrairement aux champs, les propriétés ne désignent pas les emplacements de stockage. Au lieu de cela, les propriétés ont des *accesseurs* qui spécifient les instructions exécutées lorsque leurs valeurs sont lues ou écrites.

Une propriété est déclarée comme un champ, sauf que la déclaration se termine par un accesseur Get ou un accesseur Set écrit entre les délimiteurs `{` et `}` au lieu de se terminer par un point-virgule. Une propriété qui a un accesseur get et un accesseur set est une *propriété en lecture-écriture*, une propriété qui possède uniquement un accesseur get est une *propriété en lecture seule*, et une propriété qui possède uniquement un accesseur set est une *propriété en écriture seule*.

Un accesseur get correspond à une méthode sans paramètre avec une valeur de retour du type de la propriété. Un accesseur set correspond à une méthode avec un paramètre unique nommé valeur et aucun type de retour. L'accesseur Get calcule la valeur de la propriété. L'accesseur Set fournit une nouvelle valeur pour la propriété. Lorsque la propriété est la cible d'une assignation, ou l'opérande de `++` ou `--` , l'accesseur Set est appelé. Dans les autres cas où la propriété est référencée, l'accesseur Get est appelé.

La classe `MyList<T>` déclare deux propriétés, `Count` et `Capacity`, qui sont respectivement en lecture seule et en lecture-écriture. Le code suivant est un exemple d'utilisation de ces propriétés :

```
MyList<string> names = new MyList<string>();
names.Capacity = 100;   // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

C# prend en charge les propriétés d'instance et les propriétés statiques, similaires aux champs et aux méthodes. Les propriétés statiques sont déclarées avec le modificateur static, et les propriétés d'instance sont déclarées sans.

Le ou les accesseurs d'une propriété peuvent être virtuels. Lorsqu'une déclaration de propriété inclut un modificateur `virtual`, `abstract` ou `override`, elle s'applique aux accesseurs de la propriété.

Indexeurs

Un *indexeur* est un membre qui permet l'indexation des objets de la même façon en tant que tableau. Un indexeur est déclaré comme une propriété, sauf que le nom du membre est `this`, suivi d'une liste de paramètres écrits entre les délimiteurs `[` et `]`. Les paramètres sont disponibles dans le ou les accesseurs de l'indexeur. Similaires aux propriétés, les indexeurs peuvent être en lecture-écriture, en lecture seule et en écriture seule, et les accesseurs d'un indexeur peuvent être virtuels.

La classe `MyList<T>` déclare un indexeur en lecture-écriture unique qui prend un paramètre `int`. L'indexeur rend possible l'indexation des instances `MyList<T>` avec des valeurs `int`. Exemple :

```
MyList<string> names = new MyList<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Les indexeurs peuvent être surchargés. Une classe peut déclarer plusieurs indexeurs tant que le nombre ou les types de leurs paramètres diffèrent.

Événements

Un *événement* est un membre qui permet à une classe ou un objet de fournir des notifications. Un événement est déclaré comme un champ, sauf que la déclaration comprend un `event` mot clé et que le type doit être un type délégué.

Dans une classe qui déclare un membre d'événement, l'événement se comporte de la même façon qu'un champ d'un type délégué (à condition que l'événement ne soit pas abstrait et ne déclare pas d'accesseurs). Le champ stocke une référence à un délégué qui représente les gestionnaires d'événements qui ont été ajoutés à l'événement. Si aucun gestionnaire d'événements n'est présent, le champ est `null`.

La classe `MyList<T>` déclare un membre d'événement unique appelé `Changed`, qui indique qu'un nouvel élément a été ajouté à la liste. L'événement Changed est déclenché par la méthode virtuelle `OnChanged`, qui vérifie si l'événement est `null` (ce qui signifie qu'aucun gestionnaire n'est présent). La notion de déclenchement d'un événement est équivalente précisément à l'appel du délégué représenté par l'événement. Il n'existe aucune construction de langage spéciale pour déclencher des événements.

Les clients réagissent aux événements via les *gestionnaires d'événements*. Les gestionnaires d'événements sont joints à l'aide de l'opérateur `+=` et supprimés à l'aide de l'opérateur `-=`. L'exemple suivant joint un gestionnaire d'événements à l'événement `Changed` d'un `MyList<string>`.

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

Pour les scénarios avancés où le contrôle du stockage sous-jacent d'un événement est souhaité, une déclaration d'événement peut fournir explicitement des `add` `remove` accesseurs et, qui sont similaires à l' `set` accesseur d'une propriété.

Opérateurs

Un *opérateur* est un membre qui définit la signification de l'application d'un opérateur d'expression particulière aux instances d'une classe. Trois types d'opérateurs peuvent être définis : les opérateurs unaires, les opérateurs binaires et les opérateurs de conversion. Tous les opérateurs doivent être déclarés comme `public` et `static`.

La `MyList<T>` classe déclare deux opérateurs, `operator ==` et `operator !=` . Ces opérateurs substitués donnent une nouvelle signification aux expressions qui appliquent ces opérateurs aux `MyList` instances. Plus précisément, les opérateurs définissent l'égalité de deux `MyList<T>` instances en comparant chacun des objets contenus à l'aide de leurs `Equals` méthodes. L'exemple suivant utilise l'opérateur `==` pour comparer deux instances de `MyList<int>`.

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

La première `Console.WriteLine` génère `True`, car les deux listes contiennent le même nombre d'objets avec les mêmes valeurs dans le même ordre. Si `MyList<T>` n'avait pas défini `operator ==`, la première `Console.WriteLine` aurait affiché `False`, car `a` et `b` référencent des instances `MyList<int>` différentes.

Finaliseurs

Un *finaliseur* est un membre qui implémente les actions requises pour finaliser une instance d'une classe. En règle générale, un finaliseur est nécessaire pour libérer des ressources non managées. Les finaliseurs ne peuvent pas avoir de paramètres, ils ne peuvent pas avoir de modificateurs d'accessibilité et ils ne peuvent pas être appelés explicitement. Le finaliseur pour une instance est appelé automatiquement lors du nettoyage de la mémoire (garbage collection). Pour plus d'informations, consultez l'article sur les finaliseurs.

Le récupérateur de mémoire est libre de déterminer quand collecter des objets et exécuter des finaliseurs. Plus précisément, le minutage des appels du finaliseur n'est pas déterministe, et les finaliseurs peuvent être exécutés sur n'importe quel thread. Pour ces raisons et d'autres, les classes doivent implémenter des finaliseurs uniquement lorsqu'aucune des autres solutions n'est envisageable.

L'instruction `using` fournit une meilleure approche pour la destruction d'objets.

Expressions

Les *expressions* sont construites à partir de *d'opérandes* et *d'opérateurs*. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes. Parmi les exemples d'opérateurs figurent `+`, `-`, `*`, `/` et `new`. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Quand une expression contient plusieurs opérateurs, la *priorité* des opérateurs contrôle l'ordre dans lequel ils sont évalués. Par exemple, l'expression `x + y * z` est évaluée comme `x + (y * z)`, car l'opérateur `*` a une priorité plus élevée que `+`.

Lorsqu'un opérande se produit entre deux opérateurs de même priorité, *l'associativité* des opérateurs détermine l'ordre dans lequel les opérations sont effectuées :

- À l'exception des opérateurs d'assignation et de fusion Null, tous les opérateurs binaires sont *associatifs à gauche*, ce qui signifie que les opérations sont effectuées de gauche à droite. Par exemple, `x + y + z` est évalué comme étant `(x + y) + z`.
- Les opérateurs d'assignation, les opérateurs de fusion Null `??` et `??=` l'opérateur conditionnel `?:` sont *associatifs à droite*, ce qui signifie que les opérations sont exécutées de droite à gauche. Par exemple,

`x = y = z` est évalué comme étant `x = (y = z)`.

La priorité et l'associativité peuvent être contrôlées à l'aide de parenthèses. Par exemple, `x + y * z` multiplie d'abord `y` par `z`, puis ajoute le résultat à `x`, mais `(x + y) * z` ajoute d'abord `x` et `y`, puis multiplie le résultat par `z`.

La plupart des opérateurs peuvent être *surchargés*. La surcharge d'opérateur autorise la spécification d'implémentations d'opérateurs définies par l'utilisateur pour les opérations où l'un des deux opérandes ou les deux sont d'un type classe ou struct défini par l'utilisateur.

C# met à votre disposition de nombreux opérateurs pour effectuer des opérations arithmétiques, logiques, au niveau du bit et de décalage et des comparaisons d'égalité et d'ordre .

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez Opérateurs C#.

Instructions

Les actions d'un programme sont exprimées à l'aide *d'instructions*. C# prend en charge plusieurs types d'instructions, dont plusieurs sont définies en termes d'instructions incorporées.

- Un *bloc* autorise l'écriture de plusieurs instructions dans les contextes où une seule instruction est autorisée. Un bloc se compose d'une liste d'instructions écrites entre les délimiteurs `{` et `}`.
- Les *instructions de déclaration* sont utilisées pour déclarer des variables locales et des constantes.
- Les *instructions d'expression* sont utilisées pour évaluer des expressions. Les expressions qui peuvent être utilisées en tant qu'instructions comprennent les appels de méthode, les allocations d'objet avec l'opérateur `new`, les affectations avec `=` et les opérateurs d'assignation composée, les opérations d'incrémentation et de décrémentation à l'aide des opérateurs `++` et `--` et les expressions `await`.
- Les *instructions de sélection* sont utilisées pour sélectionner un nombre d'instructions possibles pour l'exécution en fonction de la valeur d'une expression. Ce groupe contient les `if` `switch` instructions et.
- Les *instructions d'itération* servent à exécuter à plusieurs reprises une instruction incorporée. Ce groupe contient les `while` `do` instructions,, `for` et `foreach` .
- Les *instructions de saut* sont utilisées pour transférer le contrôle. Ce groupe contient les `break` `continue` instructions,, `goto` , `throw` , `return` et `yield` .
- L'instruction `try` ... `catch` est utilisée pour intercepter les exceptions qui se produisent pendant l'exécution d'un bloc, et l'instruction `try` ... `finally` permet de spécifier que le code de finalisation est toujours exécuté, qu'une exception se soit produite ou non.
- Les instructions `checked` et `unchecked` permettent de contrôler le contexte de dépassement de capacité pour les conversions et les opérations arithmétiques de type intégral.
- L'instruction `lock` est utilisée pour obtenir le verrou d'exclusion mutuelle pour un objet donné, exécuter une instruction, puis libérer le verrou.
- L'instruction `using` est utilisée pour obtenir une ressource, exécuter une instruction puis supprimer cette ressource.

La liste suivante répertorie les types d'instructions qui peuvent être utilisés :

- Déclaration de variable locale.
- Déclaration de constante locale.
- Instruction d'expression.
- `if` gestion.
- `switch` gestion.
- `while` gestion.
- `do` gestion.

- `for` gestion.
- `foreach` gestion.
- `break` gestion.
- `continue` gestion.
- `goto` gestion.
- `return` gestion.
- `yield` gestion.
- `throw` instructions et `try` instructions.
- `checked``unchecked` instructions et.
- `lock` gestion.
- `using` gestion.

PRÉCÉDENT SUIVANT

Principales zones de langue

16/12/2020 • 16 minutes to read • Edit Online

Tableaux, collections et LINQ

C# et .NET fournissent de nombreux types de collections différents. Les tableaux ont une syntaxe définie par le langage. Les types de collections génériques sont répertoriés dans l' System.Collections.Generic espace de noms. Les collections spécialisées incluent l' System.Span<T> accès à la mémoire continue sur le frame de pile et l' System.Memory<T> accès à la mémoire continue sur le tas managé. Toutes les collections, y compris les tableaux, Span<T> et Memory<T> partagent un principe d'unification pour l'itération. Vous utilisez l' System.Collections.Generic.IEnumerable<T> interface. Ce principe d'unification signifie que tous les types de collections peuvent être utilisés avec des requêtes LINQ ou d'autres algorithmes. Vous écrivez des méthodes à l'aide de IEnumerable<T> et ces algorithmes fonctionnent avec n'importe quelle collection.

Tableaux

Un * **tableau** _ est une structure de données qui contient un certain nombre de variables accessibles par le biais d'index calculés. Les variables contenues dans un tableau, également appelées *éléments* du tableau, sont toutes du même type. Ce type est appelé le *type d'élément* du tableau.

Les types tableau sont des types référence, et la déclaration d'une variable tableau réserve simplement un espace pour une référence à une instance de tableau. Les instances de tableau réelles sont créées dynamiquement au moment de l'exécution à l'aide de l' `new` opérateur. L' `new` opération spécifie la *longueur* de la nouvelle instance de tableau, qui est ensuite résolue pour la durée de vie de l'instance. Les indices des éléments d'un tableau vont de `0` à `Length - 1`. L'opérateur `new` initialise automatiquement les éléments d'un tableau à leur valeur par défaut, c'est-à-dire, par exemple, zéro pour tous les types numériques et `null` pour tous les types référence.

L'exemple suivant crée un tableau de `int` éléments, initialise le tableau et imprime le contenu du tableau.

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

Cet exemple crée et opère sur un *tableau unidimensionnel*. C# prend également en charge *les tableaux multidimensionnels*. Le nombre de dimensions d'un type tableau, également appelé *rang* du type tableau, est un plus le nombre de virgules écrites entre les crochets du type tableau. L'exemple suivant alloue respectivement des tableaux à une seule, deux et trois dimensions, respectivement.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Le tableau `a1` contient 10 éléments, le tableau `a2` en contient 50 (10 x 5) et le tableau `a3` en contient 100 (10 × 5 × 2). Le type d'élément d'un tableau peut être de n'importe quel type, y compris un type tableau. Un tableau avec des éléments d'un type tableau est parfois appelé *tableau en escalier* , car les longueurs des tableaux d'éléments ne doivent pas nécessairement être identiques. L'exemple suivant alloue un tableau de tableaux de `int` :

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La première ligne crée un tableau avec trois éléments, chacun de type `int[]` et chacun avec une valeur initiale de `null`. Les lignes suivantes initialisent ensuite les trois éléments avec des références à des instances de tableau individuelles de longueurs différentes.

L' `new` opérateur permet de spécifier les valeurs initiales des éléments du tableau à l'aide d'un *initialiseur de tableau*, qui est une liste d'expressions écrites entre les délimiteurs `{` et `}`. L'exemple suivant alloue et initialise un `int[]` avec trois éléments.

```
int[] a = new int[] { 1, 2, 3 };
```

La longueur du tableau est déduite du nombre d'expressions entre `{` et `}`. L'initialisation de tableau peut être raccourcie de manière à ce que le type de tableau n'ait pas à être retraité.

```
int[] a = { 1, 2, 3 };
```

Les deux exemples précédents sont équivalents au code suivant :

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

L' `foreach` instruction peut être utilisée pour énumérer les éléments d'une collection. Le code suivant énumère le tableau de l'exemple précédent :

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

L' `foreach` instruction utilise l' IEnumerable<T> interface, donc peut fonctionner avec n'importe quelle collection.

Interpolation de chaîne

L' *interpolation de chaîne* C# vous permet de mettre en forme des chaînes en définissant des expressions dont les résultats sont placés dans une chaîne de format. Par exemple, l'exemple suivant imprime la température d'un jour donné à partir d'un ensemble de données météorologiques :

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"    was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//     was 5 and 30.
```

Une chaîne interpolée est déclarée à l'aide du `$` jeton. L'interpolation de chaîne évalue les expressions comprises entre `{` et `}`, puis convertit le résultat en `string` et remplace le texte entre crochets par le résultat de chaîne de l'expression. `:` Dans la première expression, `{weatherData.Date:MM-DD-YYYY}` spécifie le _Format chaîne *. Dans

l'exemple précédent, il spécifie que la date doit être imprimée au format « MM-jj-aaaa ».

Critères spéciaux

Le langage C# fournit des expressions de * **critères spéciaux** pour interroger l'état d'un objet et exécuter du code en fonction de cet État. Vous pouvez inspecter les types et les valeurs des propriétés et des champs pour déterminer l'action à entreprendre. L' `switch` expression est l'expression principale pour les critères spéciaux.

Délégués et expressions lambda

Un *type délégué* représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction trouvés dans d'autres langages. Contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

L'exemple suivant déclare et utilise un type délégué nommé `Function`.

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

Une instance du type délégué `Function` peut faire référence à toute méthode qui accepte un argument `double` et retourne une valeur `double`. La `Apply` méthode applique un donné `Function` aux éléments d'un `double[]` , en retournant un `double[]` avec les résultats. Dans la méthode `Main`, `Apply` sert à appliquer trois fonctions différentes à un `double[]`.

Un délégué peut faire référence à une méthode statique (comme `Square` ou `Math.Sin` dans l'exemple précédent) ou à une méthode d'instance (comme `m.Multiply` dans l'exemple précédent). Un délégué qui référence une méthode d'instance référence aussi un objet particulier, et quand la méthode d'instance est appelée via le délégué, cet objet devient `this` dans l'appel.

Les délégués peuvent également être créés à l'aide de fonctions anonymes, qui sont des « méthodes inline » qui sont créées lorsqu'elles sont déclarées. Les fonctions anonymes peuvent voir les variables locales des méthodes

qui l'entourent. L'exemple suivant ne crée pas de classe :

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

Un délégué ne connaît pas ou ne tient pas compte de la classe de la méthode à laquelle il fait référence. Il est important que la méthode référencée ait les mêmes paramètres et le même type de retour que le délégué.

Async/await

C# prend en charge les programmes asynchrones avec deux mots clés : `async` et `await` . Vous ajoutez le `async` modificateur à une déclaration de méthode pour déclarer que la méthode est asynchrone. L' `await` opérateur indique au compilateur d'attendre de façon asynchrone qu'un résultat se termine. Le contrôle est retourné à l'appelant, et la méthode retourne une structure qui gère l'état du travail asynchrone. La structure est généralement un System.Threading.Tasks.Task<TResult> , mais peut être n'importe quel type qui prend en charge le modèle d'attente. Ces fonctionnalités vous permettent d'écrire du code qui se lit comme son homologue synchrone, mais qui s'exécute de façon asynchrone. Par exemple, le code suivant télécharge la page d'origine de Microsoft docs:

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}
```

Ce petit exemple illustre les principales fonctionnalités de la programmation asynchrone :

- La déclaration de méthode comprend le `async` modificateur.
- Corps de la méthode `await` s le retour de la `GetByteArrayAsync` méthode.
- Le type spécifié dans l' `return` instruction correspond à l'argument de type dans la `Task<T>` déclaration de la méthode. (Une méthode qui retourne un `Task` utilise `return` des instructions sans argument).

Attributs

Les types, membres et autres entités d'un programme C# prennent en charge les modificateurs qui contrôlent certains aspects de leur comportement. Par exemple, l'accessibilité d'une méthode est contrôlée à l'aide des modificateurs `public` , `protected` , `internal` et `private` . C# généralise cette fonctionnalité pour que les types d'informations déclaratives définis par l'utilisateur puissent être associés aux entités de programme et récupérés au moment de l'exécution. Les programmes spécifient ces informations déclaratives supplémentaires en définissant et en utilisant *_ attributes *.*

L'exemple suivant déclare un attribut `HelpAttribute` qui peut être placé sur des entités de programme pour fournir des liens vers leur documentation associée.

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

Toutes les classes d'attributs dérivent de la Attribute classe de base fournie par la bibliothèque .net. Les attributs peuvent être appliqués en donnant leur nom, ainsi que les éventuels arguments, à l'intérieur de crochets juste avant la déclaration associée. Si le nom d'un attribut se termine en `Attribute`, cette partie du nom peut être omise lorsque l'attribut est référencé. Par exemple, `HelpAttribute` peut être utilisé comme suit.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
    Topic = "Display")]
    public void Display(string text) { }
}
```

Cet exemple joint un `HelpAttribute` à la classe `Widget`. Il ajoute un autre `HelpAttribute` à la méthode `Display` dans la classe. Les constructeurs publics d'une classe d'attributs contrôlent les informations qui doivent être fournies lorsque l'attribut est joint à une entité de programme. Des informations supplémentaires peuvent être fournies en référençant les propriétés en lecture-écriture publiques de la classe d'attributs (comme la référence à la propriété `Topic`, précédemment).

Les métadonnées définies par les attributs peuvent être lues et manipulées par réflexion lors de l'exécution. Lorsqu'un attribut particulier est demandé à l'aide de cette technique, le constructeur de la classe d'attributs est appelé avec les informations fournies dans la source du programme, et l'instance d'attribut qui en résulte est retournée. Si des informations supplémentaires ont été fournies via les propriétés, ces propriétés sont définies sur les valeurs données avant que le renvoi de l'instance de l'attribut.

L'exemple de code suivant montre comment obtenir les `HelpAttribute` instances associées à la classe `Widget` et sa méthode `Display`.

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

En savoir plus

Vous pouvez en savoir plus sur C# en essayant l'un de nos didacticiels.

PRÉCÉDENT

Didacticiels C#

16/12/2020 • 7 minutes to read • Edit Online

Bienvenue dans les tutoriels pour C#. Ceux-ci commencent par des leçons interactives que vous pouvez exécuter dans votre navigateur. Les tutoriels suivants et d'autres plus avancés vous aident à utiliser les outils de développement .NET pour créer des programmes en C# sur votre ordinateur.

Tutoriels interactifs de présentation de C#

Si vous souhaitez commencer votre exploration au format vidéo, la série de vidéos c# 101 fournit une introduction à c#. Vous en apprendrez plus sur les concepts que vous pouvez explorer dans ces didacticiels.

Les premières leçons expliquent les concepts de C# à l'aide de petits extraits de code. Vous apprendrez les bases de la syntaxe C# et comment utiliser des types de données tels que des chaînes, des nombres et des valeurs booléennes. Tout est interactif, vous allez écrire et exécuter du code en quelques minutes. Ces premières leçons prennent pour acquis que vous n'avez aucune connaissance préalable de la programmation ou du langage C#.

Salut tout le monde

Dans le tutoriel Hello world, vous allez créer le programme le plus basique en C#. Vous explorerez le type `string` et l'utilisation de texte.

Nombres en C#

Dans le tutoriel Nombres en C#, vous allez apprendre comment les ordinateurs stockent les nombres et comment effectuer des calculs avec différents types numériques. Vous apprendrez les bases de l'arrondi et comment effectuer des calculs mathématiques à l'aide de C#. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce tutoriel part du principe que vous avez terminé la leçon Hello world.

Branches et boucles

Le tutoriel Branches et boucles explique les bases de la sélection de différents chemins d'exécution de code en fonction des valeurs stockées dans des variables. Vous allez apprendre les principes fondamentaux du flux de contrôle, sur la base duquel les programmes prennent des décisions et choisissent différentes actions. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce tutoriel part du principe que vous avez fini les leçons Hello world et Nombres en C#.

Collection de listes

La leçon Collection de listes vous présente le type Collection de listes qui stocke les séquences de données. Vous apprendrez à ajouter et à supprimer des éléments, à rechercher des éléments et à trier des listes. Vous découvrirez différents types de listes. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce tutoriel part du principe que vous avez fini les leçons listées ci-dessus.

Présentation de C# -- travailler localement

Tous les tutoriels de présentation proposés après la leçon Hello World sont disponibles dans votre propre

environnement de développement local. À la fin de chaque tutoriel, vous décidez si vous souhaitez effectuer la leçon suivante en ligne ou sur votre propre ordinateur. Vous trouverez également des liens pour vous aider à configurer votre environnement et passer au tutoriel suivant sur votre ordinateur.

Explorer les nouvelles fonctionnalités du langage C#

- Interpolation de chaîne: ce tutoriel décrit comment utiliser l'interpolation de chaîne pour créer des chaînes mises en forme en C#.
- Types de référence Nullable : montre comment utiliser des types de référence nullable pour exprimer votre intention en matière de références null.
- Mettre à jour un projet de façon à utiliser des types référence Nullables : illustre des techniques de mise à niveau d'un projet existant visant à utiliser des types référence Nullables.
- Étendre les fonctionnalités des données avec les critères spéciaux : montre comment utiliser les critères spéciaux pour étendre les types au-delà de leurs fonctionnalités principales.
- Utiliser des séquences de données à l'aide d'index et de plages: illustre une nouvelle syntaxe pratique pour accéder à des éléments uniques ou à des plages d'un conteneur de données séquentielles.

Tutoriels généraux

Les didacticiels suivants vous permettent de générer des programmes C# à l'aide de .NET Core :

- Application console : ce tutoriel présente les fonctionnalités d'E/S de la console, la structure d'une application console et les concepts de base du modèle de programmation asynchrone basé sur les tâches.
- Client REST : ce tutoriel présente les communications web, la sérialisation JSON et les fonctionnalités orientées objet du langage C#.
- Héritage dans C# et .NET : ce didacticiel présente l'héritage dans C#, notamment l'utilisation de l'héritage pour définir des classes de base, des classes de base abstraites et des classes dérivées.
- Utilisation de LINQ : ce didacticiel illustre plusieurs fonctionnalités de LINQ et les éléments de langage qui le prennent en charge.
- Utilisation d'attributs : ce tutoriel décrit comment créer et utiliser des attributs en C#.
- Le tutoriel Interpolation de chaîne vous montre comment insérer des valeurs dans une chaîne. Vous allez apprendre à créer une chaîne interpolée avec des expressions C# incorporées, et à contrôler l'apparence du texte des résultats des expressions dans la chaîne de résultat. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Présentation de C#

02/11/2020 • 7 minutes to read • Edit Online

Bienvenue dans les tutoriels de présentation de C#. Ces leçons commencent par du code interactif que vous pouvez exécuter dans votre navigateur. Vous pouvez apprendre les principes fondamentaux de C# de la série de vidéos c# 101 avant de commencer ces leçons interactives.

Les premières leçons expliquent les concepts de C# à l'aide de petits extraits de code. Vous apprendrez les bases de la syntaxe C# et comment utiliser des types de données tels que des chaînes, des nombres et des valeurs booléennes. Tout est interactif, vous allez écrire et exécuter du code en quelques minutes. Ces premières leçons prennent pour acquis que vous n'avez aucune connaissance préalable de la programmation ou du langage C#.

Vous pouvez essayer ces didacticiels dans différents environnements. Les concepts que vous allez apprendre sont les mêmes. La différence réside dans l'expérience que vous préférez :

- Dans votre navigateur, sur la plateforme docs: cette expérience incorpore une fenêtre de code C# exécutable dans les pages docs. Vous écrivez et exécutez du code C# dans le navigateur.
- Dans l'expérience Microsoft Learn. Ce parcours d'apprentissage contient plusieurs modules qui enseignent les principes fondamentaux de C#.
- Dans Jupyter sur Binder. Vous pouvez expérimenter le code C# dans un bloc-notes Jupyter sur Binder.
- Sur votre ordinateur local. Une fois que vous avez exploré en ligne, vous pouvez Télécharger le kit SDK .net Core et générer des programmes sur votre ordinateur.

Tous les tutoriels de présentation proposés après la leçon Hello World sont disponibles en ligne dans le navigateur ou dans votre propre environnement de développement local. À la fin de chaque tutoriel, vous décidez si vous souhaitez effectuer la leçon suivante en ligne ou sur votre propre ordinateur. Vous trouverez également des liens pour vous aider à configurer votre environnement et passer au tutoriel suivant sur votre ordinateur.

Salut tout le monde

Dans le tutoriel Hello world, vous allez créer le programme le plus basique en C#. Vous explorerez le type `string` et l'utilisation de texte. Vous pouvez également utiliser le chemin d'accès sur Microsoft Learn ou Jupyter sur Binder.

Nombres en C#

Dans le tutoriel Nombres en C#, vous allez apprendre comment les ordinateurs stockent les nombres et comment effectuer des calculs avec différents types numériques. Vous apprendrez les bases de l'arrondi et comment effectuer des calculs mathématiques à l'aide de C#. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce didacticiel part du principe que vous avez terminé la leçon Hello World .

Branches et boucles

Le tutoriel Branches et boucles explique les bases de la sélection de différents chemins d'exécution de code en fonction des valeurs stockées dans des variables. Vous allez apprendre les principes fondamentaux du flux de contrôle, sur la base duquel les programmes prennent des décisions et choisissent différentes actions. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce didacticiel part du principe que vous avez terminé les leçons « Hello World » et « numbers » en C# .

Collection de listes

La leçon Collection de listes vous présente le type Collection de listes qui stocke les séquences de données. Vous apprendrez à ajouter et à supprimer des éléments, à rechercher des éléments et à trier des listes. Vous découvrirez différents types de listes. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce didacticiel part du principe que vous avez terminé les leçons mentionnées ci-dessus.

Introduction aux classes

Ce didacticiel est disponible uniquement pour s'exécuter sur votre ordinateur, à l'aide de votre propre environnement de développement local et de .NET Core. Vous allez créer une application console et voir les fonctionnalités orientées objet de base qui font partie du langage C#.

Ce tutoriel prend pour acquis que vous avez terminé les tutoriels de présentation en ligne et que vous avez installé le SDK .NET Core et Visual Studio Code.

Programmation orientée objet

Ce didacticiel enseigne les concepts utilisés dans la programmation orientée objet. Vous apprendrez les concepts d' *abstraction*, d' *encapsulation*, d' *héritage*et de *polymorphisme* à l'aide d'exemples C#.

Ce didacticiel part du principe que vous avez terminé les didacticiels d'introduction en ligne et que vous avez installé Kit SDK .net Core et Visual Studio code ou Visual Studio sur votre ordinateur de développement.

Exemples LINQ 101

Cet exemple requiert l'outil dotnet-try global. Une fois que vous avez installé l'outil et que vous avez cloné les référentiel try-Samples , vous pouvez apprendre LINQ (Language Integrated Query) à l'aide d'un ensemble d'exemples 101 que vous pouvez exécuter de manière interactive. Vous pouvez explorer les différentes méthodes d'interrogation, d'exploration et de transformation des séquences de données.

Vous familiariser avec les outils de développement .NET

18/03/2020 • 3 minutes to read • Edit Online

La première étape pour exécuter un tutoriel sur votre ordinateur consiste à configurer un environnement de développement. Le tutoriel .NET Hello World en 10 minutes a des instructions pour configurer votre environnement de développement local sur Windows, Linux ou macOS.

Vous pouvez aussi installer le Kit SDK .NET Core et Visual Studio Code.

Flux de développement d'une application de base

Vous créerez des `dotnet new` applications à l'aide de la commande. Cette commande génère les fichiers et ressources nécessaires pour votre application. Les tutoriels de présentation de C# utilisent tous le type d'application `console`. Une fois que vous avez les bases, vous pouvez développer d'autres types d'applications.

Les autres commandes que vous `dotnet build` utiliserez sont pour `dotnet run` construire l'exécutable, et pour exécuter l'exécutable.

Choisir un tutoriel

Vous pouvez commencer par l'un des tutoriels suivants :

Nombres en C#

Dans le tutoriel Nombres en C#, vous allez apprendre comment les ordinateurs stockent les nombres et comment effectuer des calculs avec différents types numériques. Vous apprendrez les bases de l'arrondi et comment effectuer des calculs mathématiques à l'aide de C#.

Ce tutoriel part du principe que vous avez terminé la leçon Hello world.

Branches et boucles

Le tutoriel Branches et boucles explique les bases de la sélection de différents chemins d'exécution de code en fonction des valeurs stockées dans des variables. Vous allez apprendre les principes fondamentaux du flux de contrôle, sur la base duquel les programmes prennent des décisions et choisissent différentes actions.

Ce tutoriel part du principe que vous avez fini les leçons Hello world et Nombres en C#.

Collection de listes

La leçon Collection de listes vous présente le type Collection de listes qui stocke les séquences de données. Vous apprendrez à ajouter et à supprimer des éléments, à rechercher des éléments et à trier des listes. Vous découvrirez différents types de listes.

Ce tutoriel part du principe que vous avez fini les leçons listées ci-dessus.

Présentation des classes

Ce dernier tutoriel de présentation de C#, qui s'exécute sur votre ordinateur, fait appel à votre propre environnement de développement local et à .NET Core. Vous allez créer une application console et voir les

fonctionnalités orientées objet de base qui font partie du langage C#.

Manipuler les nombres intégraux et à virgule flottante en C#

06/05/2020 • 16 minutes to read • Edit Online

Ce tutoriel permet de découvrir de manière interactive les types numériques en C#. Vous allez écrire de petites quantités de code, puis vous compilerez et exécuterez ce code. Ce tutoriel comporte une série de leçons visant à explorer les nombres et les opérations mathématiques en C#. Ces leçons présentent les concepts de base du langage C#.

Ce tutoriel suppose de disposer d'un ordinateur utilisable pour le développement. Le didacticiel .NET Hello World en 10 minutes contient des instructions pour la configuration de votre environnement de développement local sur Windows, Linux ou MacOS. Vous trouverez une brève vue d'ensemble des commandes utilisées dans Se familiariser avec les outils de développement, avec des liens vers des informations complémentaires.

Explorer les mathématiques avec des entiers

Créez un répertoire nommé *numbers-quickstart*. Créez le répertoire actif et exécutez la commande suivante :

```
dotnet new console -n NumbersInCSharp -o .
```

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez la ligne `Console.WriteLine("Hello World!");` par le code qui suit :

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Exécutez ce code en tapant `dotnet run` dans votre fenêtre de commande.

Vous avez vu une des opérations mathématiques fondamentales avec des entiers. Le `int` type représente un **entier**, un zéro, un nombre entier positif ou négatif. Vous utilisez le symbole `+` pour effectuer une addition. Les autres opérations mathématiques courantes avec des entiers sont les suivantes :

- `-` pour la soustraction
- `*` pour la multiplication
- `/` pour la division

Commencez par explorer ces différentes opérations. Ajoutez ces lignes après la ligne qui écrit la valeur de `c` :

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

Exécutez ce code en tapant `dotnet run` dans votre fenêtre de commande.

Vous pouvez également expérimenter en écrivant plusieurs opérations mathématiques dans la même ligne, si vous le souhaitez. Essayez `c = a + b - 12 * 17;` par exemple. La combinaison de variables et de nombres constants est autorisée.

> **TIP**
> Durant votre exploration de C# (ou de tout autre langage de programmation), vous commettrez des erreurs d'écriture du code. Le **compilateur** détectera ces erreurs et vous les signalera. Si la sortie contient des messages d'erreur, vérifiez attentivement l'exemple de code ainsi que le code dans votre fenêtre pour identifier les corrections à apporter. Cet exercice vous aidera à mieux comprendre la structure du code C#.

Vous avez terminé la première étape. Avant de passer à la section suivante, déplaçons le code actuel dans une méthode distincte. Cela nous permettra de travailler plus facilement avec un nouvel exemple. Renommez votre méthode `Main``WorkingWithIntegers` et écrivez une nouvelle méthode `Main` qui appelle `WorkingWithIntegers`. Lorsque vous avez terminé, votre code doit ressembler à ceci :

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;

            // addition
            int c = a + b;
            Console.WriteLine(c);

            // subtraction
            c = a - b;
            Console.WriteLine(c);

            // multiplication
            c = a * b;
            Console.WriteLine(c);

            // division
            c = a / b;
            Console.WriteLine(c);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();
        }
    }
}
```

Explorer l'ordre des opérations

Commentez l'appel à `WorkingWithIntegers()`. La sortie est ainsi moins encombrée lorsque vous travaillez dans cette section :

```
//WorkingWithIntegers();
```

`//` démarre un **commentaire** dans C#. Les commentaires correspondent à du texte que vous voulez conserver dans votre code source sans l'exécuter en tant que code. Le compilateur ne génère pas de code exécutable à partir de commentaires.

Le langage C# définit la priorité des différentes opérations mathématiques avec à l'aide de règles cohérentes avec les règles mathématiques que vous avez apprises. La multiplication et la division ont priorité sur l'addition et la soustraction. Explorez ce concept en ajoutant le code suivant à votre méthode `Main` et en exécutant `dotnet run` :

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

La sortie montre que la multiplication est effectuée avant l'addition.

Vous pouvez appliquer un ordre de calcul différent en mettant entre parenthèses la ou les opérations à exécuter

en premier. Ajoutez les lignes suivantes et exécutez à nouveau :

```
d = (a + b) * c;
Console.WriteLine(d);
```

Pratiquez en combinant plusieurs opérations différentes. Ajoutez quelque chose ressemblant aux lignes suivantes au bas de votre méthode `Main`. Essayez `dotnet run` à nouveau.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

Vous avez peut-être observé un comportement intéressant des entiers. La division d'entiers produit toujours un résultat entier, même quand vous pensez que le résultat devrait inclure une partie décimale ou fractionnaire.

Si vous n'avez pas observé ce comportement, essayez le code qui suit à la fin de votre méthode `Main` :

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Tapez `dotnet run` à nouveau pour afficher les résultats.

Avant de continuer, prenez tout le code que vous avez écrit dans cette section et placez-le dans une nouvelle méthode. Nommez cette nouvelle méthode `OrderPrecedence`. Vous devez écrire un texte similaire à ce qui suit :

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;

            // addition
            int c = a + b;
            Console.WriteLine(c);

            // subtraction
            c = a - b;
            Console.WriteLine(c);

            // multiplication
            c = a * b;
            Console.WriteLine(c);

            // division
            c = a / b;
            Console.WriteLine(c);
        }

        static void OrderPrecedence()
        {
            int a = 5;
            int b = 4;
            int c = 2;
            int d = a + b * c;
            Console.WriteLine(d);

            d = (a + b) * c;
            Console.WriteLine(d);

            d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
            Console.WriteLine(d);

            int e = 7;
            int f = 4;
            int g = 3;
            int h = (e  + f) / g;
            Console.WriteLine(h);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();

            OrderPrecedence();

        }
    }
}
```

Explorer la précision et les limites des entiers

Ce dernier exemple a montré que la division d'entiers tronque le résultat. Vous pouvez obtenir le **reste** à l'aide de l'opérateur **modulo**, à savoir le caractère `%`. Essayez le code suivant dans votre méthode `Main` :

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

Le type d'entier C# diffère des entiers mathématiques d'une autre manière : le type `int` a des limites minimale et maximale. Ajoutez ce code à votre méthode `Main` pour voir ces limites :

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

Si un calcul produit une valeur qui dépasse ces limites, vous obtenez une condition de **dépassement négatif** ou **dépassement positif**. La réponse affichée indique la plage (d'une limite à l'autre). Ajoutez ces deux lignes à votre méthode `Main` pour voir un exemple :

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notez que la réponse est très proche de l'entier minimal (négatif). Il en est de même pour `min + 2`. L'addition a produit un **dépassement négatif** des valeurs autorisées pour les entiers. La réponse est un très grand nombre négatif, car un dépassement négatif « inclut » de la plus grande valeur d'entier possible à la plus petite.

Il existe d'autres types numériques avec une précision et des limites différentes que vous pouvez utiliser quand le type `int` ne répond pas à vos besoins. Étudions les autres types suivants.

Encore une fois, déplaçons le code que vous avez écrit dans cette section vers une autre méthode. Nommez-le `TestLimits`.

Utiliser le type double

Le type numérique `double` représente un nombre à virgule flottante double précision. Ces termes vous sont peut-être inconnus. Un nombre à **virgule flottante** est utile pour représenter des nombres non intégraux qui peuvent être très grands ou très petits en amplitude. La **double précision** est un terme relatif qui décrit le nombre de chiffres binaires utilisés pour stocker la valeur. Les nombres à **double précision** ont deux fois le nombre de chiffres binaires comme **simple précision**. Sur les ordinateurs modernes, il est plus courant d'utiliser une double précision que des nombres à simple précision. Les nombres à **précision unique** sont déclarés à l'aide du `float` mot clé. Explorons ce type double. Ajoutez le code suivant et affichez le résultat :

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Notez que la réponse inclut la partie décimale du quotient. Essayez avec une expression légèrement plus complexe utilisant des doubles :

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

La plage d'une valeur double est nettement supérieure à celle de valeurs entières. Essayez le code suivant sous ce que vous avez écrit jusque-là :

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

Ces valeurs s'affichent sous forme de notation scientifique. Le nombre à gauche du `E` est le nombre significatif. Le nombre à droite est l'exposant, en puissance de 10.

Tout comme les nombres décimaux en mathématiques, les doubles en C# peuvent présenter des erreurs d'arrondi. Exécutez le code suivant :

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

Vous savez que `0.3` la répétition n'est pas exactement identique `1/3` à celle de.

Test

Essayez d'autres calculs avec de grands nombres, des petits nombres, des multiplications `double` et des divisions à l'aide du type. Effectuez des calculs plus complexes.

Après avoir consacré plus de temps au test, placez le code que vous avez écrit dans une nouvelle méthode. Nommez cette nouvelle méthode `WorkWithDoubles`.

Utiliser des types décimaux

Vous avez vu les types numériques de base en C#, à savoir les entiers et les doubles. Il y a un autre type à apprendre : `decimal` le type. Le type `decimal` a une plage plus petite, mais une précision supérieure à celle du type `double`. Étudions cela d'un peu plus près :

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notez que la plage est plus petite que celle du type `double`. Vous pouvez observer la plus haute précision du type décimal en exécutant le code suivant :

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

Le suffixe `M` des nombres permet d'indiquer comment une constante doit utiliser le type `decimal`. Sinon, le compilateur suppose le `double` type.

> **NOTE**
> La lettre `M` a été choisie comme lettre la plus distincte entre les `double` Mots clés `decimal` et.

Notez que le calcul utilisant le type décimal a plus de chiffres à droite de la virgule décimale.

Test

Maintenant que vous avez vu les différents types numériques, écrivez du code qui calcule la surface d'un cercle avec un rayon de 2,5 centimètres. Rappelez-vous que la surface d'un cercle est le rayon au carré multiplié par PI. Conseil : .NET contient une constante pour PI, à savoir Math.PI, que vous pouvez utiliser pour cette valeur. Math.PI, comme toutes les constantes déclarées `System.Math` dans l'espace de `double` noms, est une valeur. Pour cette raison, vous devez utiliser `double` au lieu `decimal` de valeurs pour ce défi.

Vous devriez obtenir une réponse comprise entre 19 et 20. Vous pouvez vérifier votre réponse en examinant l'exemple de code terminé sur GitHub.

Si vous le voulez, essayez d'autres formules.

Vous avez terminé le guide de démarrage rapide « Nombres en C# ». Vous pouvez passer au guide de démarrage rapide Branches et boucles dans votre propre environnement de développement.

Vous pouvez en savoir plus sur les nombres en C# dans les articles suivants :

- Types numériques intégraux
- Types numériques à virgule flottante
- Conversions numériques intégrées

Découvrir la logique conditionnelle avec des instructions de branches et de boucles

14/05/2020 • 17 minutes to read • Edit Online

Ce tutoriel explique comment écrire du code qui examine des variables et modifie le chemin d'exécution en fonction de ces variables. Vous allez écrire un code en C# et afficher les résultats de la compilation et de l'exécution du code. Ce tutoriel comporte une série de leçons visant à explorer les constructions de type branches et boucles en C#. Ces leçons présentent les concepts de base du langage C#.

Ce tutoriel suppose de disposer d'un ordinateur utilisable pour le développement. Le didacticiel .NET Hello World en 10 minutes contient des instructions pour la configuration de votre environnement de développement local sur Windows, Linux ou MacOS. Vous trouverez une brève vue d'ensemble des commandes utilisées dans Se familiariser avec les outils de développement, avec des liens vers des informations complémentaires.

Prendre des décisions à l'aide de l'instruction `if`

Créez un répertoire nommé *branches-tutorial*. Créez le répertoire actif et exécutez la commande suivante :

```
dotnet new console -n BranchesAndLoops -o .
```

Cette commande crée une nouvelle application console .NET Core dans le répertoire actuel.

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez la ligne `Console.WriteLine("Hello World!");` par le code qui suit :

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Essayez ce code en tapant `dotnet run` dans la fenêtre de console. Le message « La réponse est supérieure à 10 » doit s'afficher dans votre console.

Modifiez la déclaration de `b` pour que la somme soit inférieure à 10 :

```
int b = 3;
```

Tapez `dotnet run` à nouveau. La réponse étant inférieure à 10, rien ne s'affiche. La **condition** que vous testez a une valeur false. Vous n'avez pas de code à exécuter, car vous avez uniquement écrit l'une des branches possibles pour une instruction `if` : la branche true.

> **TIP**
>
> Durant votre exploration de C# (ou de tout autre langage de programmation), vous commettrez des erreurs d'écriture du code. Le compilateur détectera et signalera les erreurs. Observez attentivement la sortie d'erreur et le code ayant généré l'erreur. L'erreur du compilateur peut en général vous aider à trouver le problème.

Le premier exemple montre la puissance de l'instruction `if` et des types booléens. Un *booléen* est une variable

qui peut avoir l'une des deux valeurs suivantes : `true` ou `false`. C# définit un type spécial, `bool`, pour les variables booléennes. L'instruction `if` vérifie la valeur d'un `bool`. Quand la valeur est `true`, l'instruction qui suit `if` s'exécute. Dans le cas contraire, elle est ignorée.

Ce processus de vérification des conditions et d'exécution d'instructions en fonction de ces conditions est puissant.

Utiliser if et else ensemble

Pour exécuter un code différent dans les branches true et false, vous créez une branche `else` qui s'exécute quand la condition a une valeur false. Essayez ceci. Ajoutez les deux dernières lignes dans le code ci-dessous à votre méthode `Main` (vous devriez déjà avoir les quatre premières lignes) :

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

L'instruction qui suit le mot clé `else` s'exécute uniquement quand la condition testée a une valeur `false`. La combinaison de `if` et `else` avec des conditions booléennes offre les hautes performances dont vous avez besoin pour traiter une condition `true` et une condition `false`.

> **IMPORTANT**
> La mise en retrait sous les instructions `if` et `else` a simplement pour but de faciliter la lecture. Le langage C# ne considère pas la mise en retrait ou les espaces blancs comme des éléments significatifs. L'instruction qui suit le mot clé `if` ou `else` sera exécutée en fonction de la condition. Tous les exemples de ce tutoriel suivent une pratique courante qui consiste à mettre en retrait les lignes en fonction du flux de contrôle des instructions.

Étant donné que la mise en retrait n'est pas significative, vous devez utiliser `{` et `}` pour indiquer quand vous souhaitez que plusieurs instructions fassent partie du bloc qui s'exécute de manière conditionnelle. Les programmeurs C# utilisent généralement les accolades pour toutes les clauses `if` et `else`. L'exemple suivant est identique à celui que vous avez créé. Modifiez votre code ci-dessus pour qu'il corresponde au code suivant :

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **TIP**
> Dans le reste de ce didacticiel, tous les exemples de code incluent les accolades, conformément aux pratiques acceptées.

Vous pouvez tester des conditions plus complexes. Ajoutez le code suivant à votre méthode `Main` après le code que vous avez écrit jusque-là :

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

Le symbole `==` teste *l'égalité*. `==` se distingue du test d'égalité d'attribution, que nous avons vu avec `a = 5`.

`&&` représente « et ». Cela signifie que les deux conditions doivent être true pour que l'instruction s'exécute dans la branche true. Ces exemples montrent également que vous pouvez avoir plusieurs instructions dans chaque branche conditionnelle, à condition de les mettre entre `{` et `}`.

Vous pouvez également utiliser `||` pour représenter « ou ». Ajoutez le code suivant après ce que vous avez écrit jusque-là :

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modifiez les valeurs de `a`, `b` et `c` et passez de `&&` à `||` et inversement. Vous comprendrez mieux comment fonctionnent les opérateurs `&&` et `||`.

Vous avez terminé la première étape. Avant de passer à la section suivante, déplaçons le code actuel dans une méthode distincte. Cela nous permettra de travailler plus facilement avec un nouvel exemple. Renommez votre méthode `Main``ExploreIf` et écrivez une nouvelle méthode `Main` qui appelle `ExploreIf`. Lorsque vous avez terminé, votre code doit ressembler à ceci :

```
using System;

namespace BranchesAndLoops
{
    class Program
    {
        static void ExploreIf()
        {
            int a = 5;
            int b = 3;
            if (a + b > 10)
            {
                Console.WriteLine("The answer is greater than 10");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
            }

            int c = 4;
            if ((a + b + c > 10) && (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("And the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("Or the first number is not greater than the second");
            }

            if ((a + b + c > 10) || (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("Or the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("And the first number is not greater than the second");
            }
        }

        static void Main(string[] args)
        {
            ExploreIf();
        }
    }
}
```

Commentez l'appel à `ExploreIf()`. La sortie est ainsi moins encombrée lorsque vous travaillez dans cette section :

```
//ExploreIf();
```

`//` démarre un **commentaire** dans C#. Les commentaires correspondent à du texte que vous voulez conserver dans votre code source sans l'exécuter en tant que code. Le compilateur ne génère pas de code exécutable à partir de commentaires.

Utiliser des boucles pour répéter des opérations

Dans cette section, vous utilisez des **boucles** pour répéter des instructions. Essayez ce code dans votre méthode

`Main` :

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

L'instruction `while` vérifie une condition et exécute l'instruction ou le bloc d'instructions qui suit `while`. Elle répète la vérification de la condition et l'exécution de ces instructions jusqu'à ce que la condition soit false.

Cet exemple contient un nouvel opérateur. `++` après la variable `counter` est l'opérateur d'**incrémentation**. Il ajoute 1 à la valeur de `counter` et stocke cette valeur dans la variable de `counter`.

> **IMPORTANT**
> Assurez-vous que la condition de boucle `while` devient false quand vous exécutez le code. Dans le cas contraire, vous allez créer une **boucle infinie** dans laquelle votre programme ne se terminera jamais. Ceci n'est pas démontré dans cet exemple, car vous devez forcer la fermeture de votre programme avec **CTRL-C** ou à l'aide d'autres moyens.

La boucle `while` teste la condition avant d'exécuter le code qui suit `while`. La boucle `do` ... `while` exécute d'abord le code, puis vérifie la condition. La boucle *do while* est illustrée dans le code suivant :

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Cette boucle `do` et la boucle antérieure `while` produisent la même sortie.

Utiliser la boucle for

La boucle **for** est couramment utilisée dans C#. Essayez ce code dans votre méthode Main() :

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

Le code précédent effectue le même travail que la `while` boucle et la `do` boucle que vous avez déjà utilisée. L'instruction `for` comprend trois parties qui contrôlent son fonctionnement.

La première partie est l' **initialiseur**: `int index = 0;` déclare qui `index` est la variable de boucle et définit sa valeur initiale sur `0` .

La partie centrale est la **condition for**: `index < 10` déclare que cette `for` boucle continue à s'exécuter tant que la valeur du compteur est inférieure à 10.

La dernière partie est l' **itérateur for**: `index++` spécifie comment modifier la variable de boucle après l'exécution du bloc après l' `for` instruction. Il spécifie ici que `index` doit être incrémenté de 1 chaque fois que le bloc s'exécute.

Expérimentez vous-même. Essayez chacune des variantes suivantes :

- Modifiez l'initialiseur pour définir le démarrage à une valeur différente.
- Modifiez la condition pour définir l'arrêt à une valeur différente.

Une fois terminé, vous allez vous-même écrire des codes pour mettre en pratique ce que vous avez appris.

Il existe une autre instruction de boucle qui n'est pas traitée dans ce didacticiel : l' `foreach` instruction. L' `foreach` instruction répète son instruction pour chaque élément d'une séquence d'éléments. La plupart du temps, elle est utilisée avec les *regroupements*. elle est donc traitée dans le didacticiel suivant.

Boucles imbriquées créées

Une `while` `do` boucle, ou `for` peut être imbriquée dans une autre boucle pour créer une matrice à l'aide de la combinaison de chaque élément de la boucle externe avec chaque élément de la boucle interne. Nous allons donc créer un ensemble de paires alphanumériques pour représenter les lignes et les colonnes.

Une seule `for` boucle peut générer les lignes :

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

Une autre boucle peut générer les colonnes :

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

Vous pouvez imbriquer une boucle à l'intérieur de l'autre pour former des paires :

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

Vous pouvez voir que la boucle externe incrémente une fois pour chaque exécution complète de la boucle interne. Inversez l'imbrication de lignes et de colonnes et observez les modifications pour vous-même.

Combiner des branches et des boucles

Maintenant que vous avez vu l'instruction `if` et la création de boucles en langage C#, vérifiez si vous pouvez écrire un code C# pour obtenir la somme de tous les entiers de 1 à 20 divisibles par 3. Voici quelques conseils :

- L'opérateur `%` vous donne le reste d'une opération de division.
- L'instruction `if` vous donne la condition pour vérifier si un nombre doit être inclus dans la somme.
- La boucle `for` peut vous aider à répéter une série d'étapes pour tous les nombres de 1 à 20.

Essayez par vous-même et vérifiez le résultat. Vous devriez obtenir 63 comme réponse. Vous pouvez voir une réponse possible en consultant le code terminé sur GitHub.

Vous avez terminé le didacticiel « Branches et boucles ».

Vous pouvez passer au tutoriel Tableaux et collections dans votre propre environnement de développement.

Pour plus d'informations sur ces concepts, consultez les articles suivants :

- Instruction if et else
- While (instruction)
- Instruction do
- Instruction for

Apprenez à gérer les collections de données en utilisant le type de liste générique

02/11/2020 • 10 minutes to read • Edit Online

Ce didacticiel propose une introduction au langage C# et présente les concepts de base de la classe List<T>.

Ce tutoriel suppose de disposer d'un ordinateur utilisable pour le développement. Le didacticiel .NET Hello World en 10 minutes contient des instructions pour la configuration de votre environnement de développement local sur Windows, Linux ou MacOS. Vous trouverez une brève vue d'ensemble des commandes utilisées disponible dans Se familiariser avec les outils de développement, avec des liens vers des informations complémentaires.

Exemple de liste de base

Créez un répertoire nommé *list-tutorial*. Faites-en le répertoire actuel et exécutez `dotnet new console`.

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez le code existant par le suivant :

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Remplacez `<name>` par votre nom. Enregistrez *Program.cs*. Tapez `dotnet run` dans votre fenêtre de console pour effectuer un essai.

Vous avez créé une liste de chaînes, ajouté trois noms à cette liste et affiché les noms en majuscules. Vous utilisez des concepts que vous avez appris dans les tutoriels précédents pour lire la liste en boucle.

Le code permettant d'afficher les noms utilise la fonctionnalité d'interpolation de chaîne. Quand vous faites précéder une `string` du caractère `$`, vous pouvez incorporer le code C# dans la déclaration de chaîne. La chaîne réelle remplace ce code C# par la valeur qu'elle génère. Dans cet exemple, elle remplace `{name.ToUpper()}` par chaque nom, converti en majuscules, car vous avez appelé la méthode ToUpper.

Continuons notre exploration.

Modifier le contenu de la liste

La collection que vous avez créée utilise le type List<T>. Ce type stocke les séquences d'éléments. Vous spécifiez le type des éléments entre crochets angulaires.

Un aspect important du type List<T> est qu'il peut augmenter ou diminuer, ce qui vous permet d'ajouter ou de

supprimer des éléments. Ajoutez ce code avant de fermer le `}` dans la méthode `Main` :

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Vous avez ajouté deux noms supplémentaires à la fin de la liste. Vous en avez également supprimé un. Enregistrez le fichier et tapez `dotnet run` pour effectuer un essai.

La List<T> vous permet en outre de faire référence à des éléments individuels par **index**. Vous placez l'index entre les jetons `[` et `]` après le nom de la liste. C# utilise la valeur 0 pour le premier index. Ajoutez le code directement sous le code que vous venez d'ajouter et effectuez un essai :

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

Vous ne pouvez pas accéder à un index au-delà de la fin de la liste. Rappelez-vous que les index commencent à partir de 0, de sorte que le plus grand index valide correspond au nombre d'éléments de la liste moins un. Vous pouvez vérifier la longueur de la liste à l'aide de la propriété Count. Ajoutez le code suivant à la fin de la méthode Main :

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Enregistrez le fichier et tapez de nouveau `dotnet run` pour afficher les résultats.

Trier les listes et y effectuer des recherches

Nos exemples utilisent des listes relativement petites, mais vos applications peuvent souvent générer des listes contenant beaucoup plus d'éléments, qui se comptent parfois même en milliers. Pour rechercher des éléments dans ces collections plus volumineuses, vous devez rechercher différents éléments dans la liste. La méthode IndexOf recherche un élément et retourne l'index de ce dernier. Si l'élément n'est pas dans la liste, `IndexOf` retourne `-1` . Ajoutez ce code en bas de votre méthode `Main` :

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

Il est également possible de trier les éléments de la liste. La Sort méthode trie tous les éléments de la liste dans l'ordre normal (par ordre alphabétique pour les chaînes). Ajoutez ce code en bas de notre méthode `Main` :

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Enregistrez le fichier et tapez `dotnet run` pour tester cette dernière version.

Avant de passer à la section suivante, déplaçons le code actuel dans une méthode distincte. Cela nous permettra de travailler plus facilement avec un nouvel exemple. Renommez votre méthode `Main``WorkingWithStrings` et écrivez une nouvelle méthode `Main` qui appelle `WorkingWithStrings`. Lorsque vous avez terminé, votre code doit ressembler à ceci :

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkingWithStrings();
        }

        static void WorkingWithStrings()
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine();
            names.Add("Maria");
            names.Add("Bill");
            names.Remove("Ana");
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine($"My name is {names[0]}");
            Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

            Console.WriteLine($"The list has {names.Count} people in it");

            var index = names.IndexOf("Felipe");
            if (index == -1)
            {
                Console.WriteLine($"When an item is not found, IndexOf returns {index}");
            }
            else
            {
                Console.WriteLine($"The name {names[index]} is at index {index}");
            }

            index = names.IndexOf("Not Found");
            if (index == -1)
            {
                Console.WriteLine($"When an item is not found, IndexOf returns {index}");
            }
            else
            {
                Console.WriteLine($"The name {names[index]} is at index {index}");

            }

            names.Sort();
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Listes d’autres types

Vous avez jusqu'à présent utilisé le type `string` dans les listes. Nous allons maintenant créer une List<T> en utilisant un autre type. Commençons par créer un ensemble de nombres.

Ajoutez le code suivant en bas de votre nouvelle méthode `Main` :

```
var fibonacciNumbers = new List<int> {1, 1};
```

Cela crée une liste d'entiers et définit les deux premiers entiers sur la valeur 1. Il s'agit des deux premières valeurs d'une *séquence Fibonacci* (séquence de nombres). Chaque nombre Fibonacci suivant correspond à la somme des deux nombres précédents. Ajoutez le code suivant :

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Enregistrez le fichier et tapez `dotnet run` pour afficher les résultats.

TIP

Pour vous concentrer uniquement sur cette section, vous pouvez commenter le code qui appelle `WorkingWithStrings();` . Insérez simplement deux caractères `/` devant l'appel comme suit : `// WorkingWithStrings();`.

Problème

Vérifiez si vous pouvez mettre en pratique certains des concepts appris ici et dans d'autres leçons antérieures. Approfondissez à partir de ce que vous avez créé jusqu'à présent avec les nombres de Fibonacci. Essayez d'écrire le code pour générer les 20 premiers nombres de la séquence. (Astuce, le 20ème nombre Fibonacci est 6765.)

Terminer le test

Vous pouvez voir un exemple de solution en consultant l'exemple de code terminé sur GitHub.

À chaque itération de la boucle, vous sélectionnez les deux derniers entiers de la liste, les additionner et ajoutez la valeur obtenue à la liste. La boucle se répète jusqu'à ce que vous ayez ajouté 20 éléments à la liste.

Félicitations, vous avez terminé ce didacticiel sur les listes. Vous pouvez passer au tutoriel Introduction aux classes dans votre propre environnement de développement.

Vous pouvez en savoir plus sur l'utilisation du `List` type dans l'article notions de base de .net sur les Collections. Vous allez également en découvrir plus sur de nombreux autres types de collection.

Explorez la programmation orientée objet avec des classes et des objets

02/11/2020 • 17 minutes to read • Edit Online

Ce tutoriel suppose que vous disposez d'un ordinateur que vous pouvez utiliser pour le développement. Le didacticiel .NET Hello World en 10 minutes contient des instructions pour la configuration de votre environnement de développement local sur Windows, Linux ou MacOS. Vous trouverez une brève vue d'ensemble des commandes utilisées dans Se familiariser avec les outils de développement, avec des liens vers des informations complémentaires.

Créer une application

Dans une fenêtre de terminal, créez un répertoire nommé *classes*. Vous y créerez votre application. Sélectionnez ce répertoire et tapez `dotnet new console` dans la fenêtre de console. Cette commande crée votre application. Ouvrez *Program.cs*. Il doit se présenter comme suit :

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Dans ce tutoriel, vous allez créer des types qui représentent un compte bancaire. Les développeurs définissent généralement chaque classe dans un fichier texte différent. Cela simplifie la gestion au fur et à mesure qu'un programme augmente en taille. Créez un fichier nommé *BankAccount.cs* dans le répertoire *classes*.

Ce fichier contient la définition d'un *compte bancaire*. La programmation orientée objet organise le code en créant des types sous la forme de *classes*. Ces classes contiennent le code qui représente une entité spécifique. La classe `BankAccount` représente un compte bancaire. Le code implémente des opérations spécifiques à travers des méthodes et des propriétés. Dans ce tutoriel, le compte bancaire prend en charge le comportement suivant :

1. Il contient un numéro à 10 chiffres qui identifie le compte bancaire de manière unique.
2. Il contient une chaîne qui stocke le nom du ou des détenteurs.
3. Le solde peut être récupéré.
4. Il accepte les dépôts.
5. Il accepte les retraits.
6. Le solde initial doit être positif.
7. Les retraits ne peuvent pas générer un solde négatif.

Définir le type de compte bancaire

Vous pouvez commencer par créer les éléments de base d'une classe définissant ce comportement. Créez un nouveau fichier à l'aide de la commande _*file : New**. Nommez-le *BankAccount.cs*. Ajoutez le code suivant à

votre fichier *BankAccount.cs* :

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Avant de poursuivre, examinons ce que vous venez de créer. La déclaration `namespace` permet d'organiser logiquement votre code. Ce tutoriel étant relativement petit, vous allez placer tout le code dans un même espace de noms.

`public class BankAccount` définit la classe ou le type que vous créez. Tout ce qui se trouve à l'intérieur du `{` et `}` qui suit la déclaration de classe définit l'État et le comportement de la classe. Il y a cinq**membres*** de la `BankAccount` classe. Les trois premières sont des *Propriétés*. Les propriétés sont des éléments de données qui peuvent avoir un code qui applique la validation ou d'autres règles. Les deux derniers sont des *méthodes*. Les méthodes sont des blocs de code qui effectuent une fonction unique. La lecture des noms de chacun des membres doit fournir suffisamment d'informations pour vous permettre (ou à tout autre développeur) de comprendre ce que fait la classe.

Ouvrir un nouveau compte

La première fonctionnalité à implémenter est l'ouverture d'un compte bancaire. Quand un client ouvre un compte, il doit fournir un solde initial, ainsi que des informations sur le ou les détenteurs du compte.

La création d'un nouvel objet du `BankAccount` type signifie la définition d'un *constructeur* qui assigne ces valeurs. Un *constructeur* est un membre qui porte le même nom que la classe. Il est utilisé pour initialiser des objets de ce type de classe. Ajoutez le constructeur suivant au `BankAccount` type. Placez le code suivant au-dessus de la déclaration de `MakeDeposit` :

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Les constructeurs sont appelés lorsque vous créez un objet à l'aide de `new` . Remplacez la ligne `Console.WriteLine("Hello World!");` dans _Program. cs * par le code suivant (remplacez `<name>` par votre nom) :

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial
balance.");
```

Exécutons ce que vous avez créé jusqu'à présent. Si vous utilisez Visual Studio, sélectionnez exécuter **sans débogage** dans le menu **exécuter** . Si vous utilisez une ligne de commande, tapez `dotnet run` le répertoire dans lequel vous avez créé votre projet.

Avez-vous remarqué que le numéro de compte est vide ? L'heure est venue de traiter ce point. Le numéro de compte doit être assigné une fois l'objet construit. Mais ce ne devrait pas être à l'appelant de le créer. Le code de la classe `BankAccount` devrait savoir comment assigner de nouveaux numéros de compte. Un moyen simple de le faire consiste à commencer par un numéro à 10 chiffres. Incrémentez-le chaque fois qu'un compte est créé. Enfin, stockez le numéro de compte actuel quand un objet est construit.

Ajoutez une déclaration de membre à la `BankAccount` classe. Placez la ligne de code suivante après l'accolade ouvrante `{` au début de la `BankAccount` classe :

```
private static int accountNumberSeed = 1234567890;
```

Il s'agit d'un membre de données. Celui-ci est `private`, ce qui signifie qu'il est uniquement accessible par code dans la classe `BankAccount`. C'est un moyen de séparer les responsabilités publiques (comme la présence d'un numéro de compte) de l'implémentation privée (comment les numéros de compte sont générés). Il est également `static`, ce qui signifie qu'il est partagé par toutes les objets `BankAccount`. La valeur d'une variable non statique est unique pour chaque instance de l'objet `BankAccount`. Ajoutez les deux lignes suivantes au constructeur pour assigner le numéro de compte. Placez-les après la ligne qui indique `this.Balance = initialBalance` :

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Tapez `dotnet run` pour afficher les résultats.

Créer des dépôts et des retraits

La classe de votre compte bancaire doit accepter les dépôts et les retraits pour fonctionner correctement. Nous allons implémenter des dépôts et des retraits en créant un journal de chaque transaction du compte. Ce dernier présente plusieurs avantages par rapport à la simple mise à jour du solde à chaque transaction. L'historique peut être utilisé pour auditer toutes les transactions et gérer les soldes au quotidien. En calculant le solde à partir de l'historique de toutes les transactions lorsque nécessaire, toutes les erreurs d'une même transaction qui sont résolues apparaîtront correctement dans le solde dans le calcul suivant.

Commençons par créer un type pour représenter une transaction. Il s'agit d'un type simple qui n'a aucune responsabilité. Il a besoin de quelques propriétés. Créez un fichier nommé *Transaction.cs*. Ajoutez-lui le code suivant :

```
using System;

namespace classes
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Notes { get; }

        public Transaction(decimal amount, DateTime date, string note)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

Nous allons maintenant ajouter une List<T> d'objets `Transaction` à la classe `BankAccount`. Ajoutez la déclaration suivante après le constructeur dans votre fichier *BankAccount.cs* :

```
private List<Transaction> allTransactions = new List<Transaction>();
```

Avec la classe List<T>, vous devez importer un espace de noms différent. Ajoutez le code suivant au début de *BankAccount.cs* :

```
using System.Collections.Generic;
```

À présent, nous allons calculer correctement `Balance` . Vous pouvez trouver le solde actuel en additionnant les valeurs de toutes les transactions. Comme le code est actuellement, vous pouvez uniquement obtenir le solde initial du compte. vous devrez donc mettre à jour la `Balance` propriété. Remplacez la ligne `public decimal Balance { get; }` dans *BankAccount.cs* par le code suivant :

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

Cet exemple montre un aspect important des**Propriétés*** . Vous calculez à présent le solde quand un autre programmeur en demande la valeur. Votre calcul énumère toutes les transactions et retourne la somme en tant que solde actuel.

Implémentez ensuite les méthodes `MakeDeposit` et `MakeWithdrawal`. Ces méthodes appliquent les deux règles finales suivantes : le solde initial doit être positif et un retrait ne doit pas générer de solde négatif.

Cela introduit le concept d' *exceptions*. Le moyen typique d'indiquer qu'une méthode ne peut pas effectuer correctement son travail consiste à lever une exception. Le type d'exception et le message associé décrivent

l'erreur. Ici, la méthode `MakeDeposit` lève une exception si le montant du dépôt est négatif. La `MakeWithdrawal` méthode lève une exception si le montant du retrait est négatif ou si l'application du retrait entraîne un solde négatif. Ajoutez le code suivant après la déclaration de la `allTransactions` liste :

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

L' `throw` instruction _*lève*une exception. L'exécution du bloc actuel se termine et le contrôle est transféré au premier bloc correspondant `catch` trouvé dans la pile des appels. Vous ajouterez un bloc `catch` pour tester ce code un peu plus tard.

Le constructeur devrait obtenir une modification lui permettant d'ajouter une transaction initiale au lieu de mettre directement le solde à jour. Étant donné que vous avez déjà écrit la méthode `MakeDeposit`, appelez-la à partir de votre constructeur. Le constructeur terminé doit être semblable à ce qui suit :

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now est une propriété qui retourne la date et l'heure actuelles. Pour tester cela, ajoutez quelques dépôts et retraits dans votre `Main` méthode, à la suite du code qui crée un `BankAccount` :

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Ensuite, testez l'interception des conditions d'erreur en essayant de créer un compte avec un solde négatif. Ajoutez le code suivant après le code précédent que vous venez d'ajouter :

```
// Test that the initial balances must be positive.
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

Vous utilisez les `try` `catch` instructions et pour marquer un bloc de code qui peut lever des exceptions et intercepter les erreurs que vous attendez. Vous pouvez utiliser la même technique pour tester le code qui lève une exception pour un solde négatif. Ajoutez le code suivant à la fin de votre `Main` méthode :

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Enregistrez le fichier et tapez `dotnet run` pour effectuer un essai.

Test : consigner toutes les transactions

Pour terminer ce tutoriel, vous pouvez écrire la méthode `GetAccountHistory` qui crée une `string` pour l'historique des transactions. Ajoutez cette méthode au type `BankAccount` :

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

Cette méthode utilise la classe StringBuilder pour mettre en forme une chaîne contenant une ligne par transaction. Vous avez vu le code de mise en forme de chaîne précédemment dans ces tutoriels. Vous pouvez observer le nouveau caractère `\t`. Celui-ci insère une tabulation pour mettre en forme la sortie.

Ajoutez cette ligne pour effectuer un essai dans *Program.cs* :

```
Console.WriteLine(account.GetAccountHistory());
```

Exécutez votre programme pour voir les résultats.

Étapes suivantes

Si vous vous êtes bloqué, vous pouvez voir la source de ce didacticiel dans notre référentiel GitHub.

Vous pouvez poursuivre le didacticiel sur la programmation orientée objet .

Pour plus d'informations sur ces concepts, consultez les articles suivants :

- Instruction if et else
- While (instruction)
- Instruction do
- Instruction for

Programmation Object-Oriented (C#)

16/12/2020 • 21 minutes to read • Edit Online

C# est un langage orienté objet. Quatre des techniques clés utilisées dans la programmation orientée objet sont les suivantes :

- L' *abstraction* signifie masquer les détails inutiles des consommateurs de type.
- L'*encapsulation* signifie qu'un groupe de propriétés, méthodes et autres membres corrélés est traité comme une unité ou un objet unique.
- L'*héritage* décrit la possibilité de créer des classes à partir d'une classe existante.
- Le *polymorphisme* signifie que plusieurs classes peuvent être utilisées de manière interchangeable, même si chacune des classes implémente les mêmes propriétés ou méthodes de manière différente.

Dans le didacticiel précédent, Introduction aux classes , vous avez vu l' *abstraction* et l' *encapsulation*. La `BankAccount` classe a fourni une abstraction pour le concept d'un compte bancaire. Vous pouvez modifier son implémentation sans affecter le code qui a utilisé la `BankAccount` classe. Les `BankAccount` classes et `Transaction` fournissent toutes deux l'encapsulation des composants nécessaires à la description de ces concepts dans le code.

Dans ce didacticiel, vous allez étendre cette application pour utiliser *l'héritage* et le *polymorphisme* pour ajouter de nouvelles fonctionnalités. Vous ajouterez également des fonctionnalités à la `BankAccount` classe, en tirant parti des techniques d' *abstraction* et d' *encapsulation* que vous avez apprises dans le didacticiel précédent.

Créer différents types de comptes

Une fois ce programme créé, vous recevez des demandes d'ajout de fonctionnalités. Il fonctionne très bien dans le cas où il n'existe qu'un seul type de compte bancaire. Au fil du temps, des besoins changent et des types de compte associés sont demandés :

- Un compte d'intérêt qui accumule des intérêts à la fin de chaque mois.
- Une ligne de crédit qui peut avoir un solde négatif, mais lorsqu'il y a un solde, des frais sont facturés chaque mois.
- Un compte de carte cadeau prépayé qui commence par un dépôt unique et qui ne peut être payé. Il peut être rempli une fois au début de chaque mois.

Tous ces comptes différents sont similaires à ceux de `BankAccount` la classe définie dans le didacticiel précédent. Vous pouvez copier ce code, renommer les classes et apporter des modifications. Cette technique peut fonctionner à long terme, mais elle serait plus de travail au fil du temps. Toutes les modifications sont copiées dans toutes les classes affectées.

Au lieu de cela, vous pouvez créer des types de comptes bancaires qui héritent des méthodes et des données de la `BankAccount` classe créée dans le didacticiel précédent. Ces nouvelles classes peuvent étendre la `BankAccount` classe avec le comportement spécifique requis pour chaque type :

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Chacune de ces classes *hérite* du comportement partagé de sa *classe de base* partagée, la `BankAccount` classe. Écrivez les implémentations pour les fonctionnalités nouvelles et différentes dans chacune des *classes dérivées*. Ces classes dérivées ont déjà tout le comportement défini dans la `BankAccount` classe.

Il est recommandé de créer chaque nouvelle classe dans un fichier source différent. Dans Visual Studio, vous pouvez cliquer avec le bouton droit sur le projet, puis sélectionner *Ajouter une classe* pour ajouter une nouvelle classe dans un nouveau fichier. Dans Visual Studio code, sélectionnez *fichier* , puis *nouveau* pour créer un nouveau fichier source. Dans l'un ou l'autre des outils, nommez le fichier pour qu'il corresponde à la classe : *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* et *GiftCardAccount.cs*.

Lorsque vous créez les classes comme indiqué dans l'exemple précédent, vous découvrirez qu'aucune de vos classes dérivées n'est compilée. Un constructeur est chargé d'initialiser un objet. Un constructeur de classe dérivée doit initialiser la classe dérivée et fournir des instructions sur la façon d'initialiser l'objet de classe de base inclus dans la classe dérivée. L'initialisation appropriée se produit normalement sans code supplémentaire. La `BankAccount` classe déclare un constructeur public avec la signature suivante :

```
public BankAccount(string name, decimal initialBalance)
```

Le compilateur ne génère pas de constructeur par défaut lorsque vous définissez un constructeur vous-même. Cela signifie que chaque classe dérivée doit appeler explicitement ce constructeur. Vous déclarez un constructeur qui peut passer des arguments au constructeur de classe de base. Le code suivant illustre le constructeur pour `InterestEarningAccount` :

```
public InterestEarningAccount(string name, decimal initialBalance) : base(name, initialBalance)
{
}
```

Les paramètres de ce nouveau constructeur correspondent au type de paramètre et aux noms du constructeur de classe de base. Vous utilisez la `: base()` syntaxe pour indiquer un appel à un constructeur de classe de base. Certaines classes définissent plusieurs constructeurs, et cette syntaxe vous permet de choisir le constructeur de classe de base que vous appelez. Une fois que vous avez mis à jour les constructeurs, vous pouvez développer le code de chacune des classes dérivées. La configuration requise pour les nouvelles classes peut être indiquée comme suit :

- Compte de bénéfices intéressants :
  - Obtient un crédit de 2% du solde de fin de mois.
- Une ligne de crédit :
  - Peut avoir un solde négatif, mais pas une valeur absolue supérieure à la limite de crédit.
  - Entraînera un montant de frais pour les intérêts chaque mois où le solde de fin de mois n'est pas égal à 0.
  - Est facturé pour chaque retrait qui dépasse la limite de crédit.
- Un compte de carte cadeau :

- Peut être rempli avec un montant spécifié une fois par mois, le dernier jour du mois.

Vous pouvez voir que ces trois types de compte ont une action qui a lieu à la fin de chaque mois. Toutefois, chaque type de compte effectue des tâches différentes. Vous utilisez le *polymorphisme* pour implémenter ce code. Créez une `virtual` méthode unique dans la `BankAccount` classe :

```
public virtual void PerformMonthEndTransactions() { }
```

Le code précédent montre comment utiliser le `virtual` mot clé pour déclarer une méthode dans la classe de base qu'une classe dérivée peut fournir une implémentation différente pour. Une `virtual` méthode est une méthode dans laquelle toute classe dérivée peut choisir de réimplémenter. Les classes dérivées utilisent le `override` mot clé pour définir la nouvelle implémentation. En général, vous faites référence à ceci comme « substitution de l'implémentation de la classe de base ». Le `virtual` mot clé spécifie que les classes dérivées peuvent substituer le comportement. Vous pouvez également déclarer `abstract` des méthodes dans lesquelles les classes dérivées doivent remplacer le comportement. La classe de base ne fournit pas d'implémentation pour une `abstract` méthode. Ensuite, vous devez définir l'implémentation de deux des nouvelles classes que vous avez créées. Commencez par le `InterestEarningAccount` :

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Ajoutez le code suivant au `LineOfCreditAccount` . Le code inverse le solde pour calculer un montant positif qui est retiré du compte :

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

La `GiftCardAccount` classe a besoin de deux modifications pour implémenter ses fonctionnalités de fin de mois. Tout d'abord, modifiez le constructeur pour inclure une quantité facultative à ajouter chaque mois :

```
private decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit = 0) : base(name,
initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

Le constructeur fournit une valeur par défaut pour la `monthlyDeposit` valeur. ainsi, les appelants peuvent omettre un `0` pour aucun dépôt mensuel. Ensuite, remplacez la `PerformMonthEndTransactions` méthode pour ajouter le dépôt mensuel, s'il a été défini sur une valeur différente de zéro dans le constructeur :

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

Le remplacement applique l'ensemble de dépôts mensuels dans le constructeur. Ajoutez le code suivant à la `Main` méthode pour tester ces modifications pour le `GiftCardAccount` et le `InterestEarningAccount` :

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Vérifiez les résultats. Ajoutez maintenant un ensemble similaire de code de test pour le `LineOfCreditAccount` :

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Lorsque vous ajoutez le code précédent et exécutez le programme, un message semblable à l'erreur suivante s'affiche :

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter
'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in BankAccount.cs:line
42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

> **NOTE**
>
> La sortie réelle comprend le chemin d'accès complet au dossier contenant le projet. Les noms de dossiers ont été omis par souci de concision. En outre, selon le format de votre code, les numéros de ligne peuvent être légèrement différents.

Ce code échoue car le `BankAccount` suppose que le solde initial doit être supérieur à 0. Une autre hypothèse intégrée à la `BankAccount` classe est que le solde ne peut pas être négatif. Au lieu de cela, tout retrait qui dédessine le compte est rejeté. Ces deux hypothèses doivent être modifiées. Le compte de la ligne de crédit commence à 0 et

est généralement un solde négatif. En outre, si un client emprunte trop d'argent, il est facturé. La transaction est acceptée, mais elle coûte juste plus cher. La première règle peut être implémentée en ajoutant un argument facultatif au `BankAccount` constructeur qui spécifie le solde minimal. La valeur par défaut est `0`. La deuxième règle requiert un mécanisme qui permet aux classes dérivées de modifier l'algorithme par défaut. Dans un sens, la classe de base « demande » le type dérivé qui doit se produire lorsqu'il y a un surplus. Le comportement par défaut consiste à rejeter la transaction en levant une exception.

Commençons par ajouter un deuxième constructeur qui comprend un paramètre facultatif `minimumBalance` . Ce nouveau constructeur effectue toutes les actions effectuées par le constructeur existant. En outre, il définit la propriété solde minimal. Vous pouvez copier le corps du constructeur existant. mais cela signifie que deux emplacements doivent être modifiés à l'avenir. Au lieu de cela, vous pouvez utiliser le *chaînage de constructeur* pour qu'un constructeur en appelle un autre. Le code suivant montre les deux constructeurs et le nouveau champ supplémentaire :

```
private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

Le code précédent illustre deux nouvelles techniques. Tout d'abord, le `minimumBalance` champ est marqué comme `readonly` . Cela signifie que la valeur ne peut pas être modifiée après la construction de l'objet. Une fois `BankAccount` créé, le `minimumBalance` ne peut pas changer. Deuxièmement, le constructeur qui accepte deux paramètres utilise `: this(name, initialBalance, 0) { }` comme implémentation. L' `: this()` expression appelle l'autre constructeur, celle avec trois paramètres. Cette technique vous permet d'avoir une seule implémentation pour l'initialisation d'un objet même si le code client peut choisir l'un des nombreux constructeurs.

Cette implémentation appelle `MakeDeposit` uniquement si le solde initial est supérieur à `0` . Cela préserve la règle selon laquelle les dépôts doivent être positifs, tout en permettant au compte de crédit de s'ouvrir avec un `0` solde.

Maintenant que la `BankAccount` classe a un champ en lecture seule pour le solde minimal, la modification finale consiste à remplacer le code en dur par `0` `minimumBalance` dans la `MakeWithdrawal` méthode :

```
if (Balance - amount < minimumBalance)
```

Après l'extension de la `BankAccount` classe, vous pouvez modifier le `LineOfCreditAccount` constructeur pour appeler le nouveau constructeur de base, comme indiqué dans le code suivant :

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name,
initialBalance, -creditLimit)
{
}
```

Notez que le `LineOfCreditAccount` constructeur modifie le signe du `creditLimit` paramètre afin qu'il corresponde à la signification du `minimumBalance` paramètre.

Autres règles de prébrouillon

La dernière fonctionnalité à ajouter permet au `LineOfCreditAccount` de payer une taxe pour dépasser la limite de crédit au lieu de refuser la transaction.

Une technique consiste à définir une fonction virtuelle où vous implémentez le comportement requis. La `BankAccount` classe refactorise la `MakeWithdrawal` méthode en deux méthodes. La nouvelle méthode effectue l'action spécifiée lorsque le retrait prend le solde inférieur au minimum. La `MakeWithdrawal` méthode existante a le code suivant :

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

Remplacez-le par le code suivant :

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    else
    {
        return default;
    }
}
```

La méthode ajoutée est `protected` , ce qui signifie qu'elle ne peut être appelée qu'à partir de classes dérivées. Cette déclaration empêche d'autres clients d'appeler la méthode. Cela permet également `virtual` aux classes dérivées de modifier le comportement. Le type de retour est `Transaction?` . L' `?` annotation indique que la méthode peut retourner `null` . Ajoutez l'implémentation suivante dans le `LineOfCreditAccount` pour payer des frais lorsque la limite de retrait est dépassée :

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

Le remplacement retourne une transaction de frais lorsque le compte est redessiné. Si le retrait n'excède pas la limite, la méthode retourne une `null` transaction. Cela indique qu'il n'y a aucun frais. Testez ces modifications en ajoutant le code suivant à votre `Main` méthode dans la `Program` classe :

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Exécutez le programme et vérifiez les résultats.

Résumé

Si vous vous êtes bloqué, vous pouvez voir la source de ce didacticiel dans notre référentiel GitHub.

Ce didacticiel a démontré de nombreuses techniques utilisées dans la programmation de Object-Oriented :

- Vous avez utilisé l' *abstraction* quand vous avez conservé de nombreux détails `private` dans chaque classe.
- Vous avez utilisé l' *encapsulation* lorsque vous avez défini des classes pour chacun des différents types de compte. Ces classes décrivent le comportement de ce type de compte.
- Vous avez utilisé l' *héritage* lorsque vous tirez parti de l'implémentation déjà créée dans la `BankAccount` classe pour enregistrer le code.
- Vous avez utilisé le *polymorphisme* quand vous avez créé `virtual` des méthodes que les classes dérivées peuvent substituer pour créer un comportement spécifique pour ce type de compte.

Félicitations, vous avez terminé tout notre introduction aux didacticiels C#. Pour en savoir plus, consultez les didacticielssuivants.

Créer des types d'enregistrements

16/12/2020 • 21 minutes to read • Edit Online

C# 9 introduit les *enregistrements*, un nouveau type de référence que vous pouvez créer à la place des classes ou des structs. Les enregistrements sont distincts des classes dans les types d'enregistrements utilisant l' *égalité basée sur les valeurs*. Deux variables d'un type d'enregistrement sont égales si les définitions de type d'enregistrement sont identiques et, si pour chaque champ, les valeurs des deux enregistrements sont égales. Deux variables d'un type de classe sont égales si les objets référencés sont du même type de classe et que les variables font référence au même objet. L'égalité basée sur la valeur implique d'autres fonctionnalités que vous souhaiterez probablement dans les types d'enregistrements. Le compilateur génère un grand nombre de ces membres lorsque vous déclarez un `record` au lieu d'un `class` .

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Déterminez si vous devez déclarer un `class` ou un `record` .
- Déclarez les types d'enregistrements et les types d'enregistrements positionnels.
- Remplacez les méthodes générées par le compilateur dans les enregistrements par vos méthodes.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 5 ou une version ultérieure, y compris le compilateur C# 9,0 ou version ultérieure. Le compilateur C# 9,0 est disponible à partir de Visual Studio 2019 version 16,8 ou du Kit de développement logiciel (SDK) .net 5,0.

Caractéristiques des enregistrements

Vous définissez un *enregistrement* en déclarant un type avec le `record` mot clé, au lieu du `class` `struct` mot clé ou. Un enregistrement est un type référence et suit la sémantique d'égalité basée sur les valeurs. Pour appliquer la sémantique de valeur, le compilateur génère plusieurs méthodes pour votre type d'enregistrement :

- Substitution de Object.Equals(Object) .
- Méthode virtuelle `Equals` dont le paramètre est le type d'enregistrement.
- Substitution de Object.GetHashCode() .
- Méthodes pour `operator ==` et `operator !=` .
- Les types d'enregistrements implémentent System.IEquatable<T> .

En outre, les enregistrements fournissent une substitution de Object.ToString() . Le compilateur synthétise les méthodes pour afficher les enregistrements à l'aide de Object.ToString() . Vous allez explorer ces membres au fur et à mesure que vous écrirez le code pour ce didacticiel. Enregistre les `with` expressions de prise en charge pour permettre une mutation non destructrice des enregistrements.

Vous pouvez également déclarer des *enregistrements positionnels* à l'aide d'une syntaxe plus concise. Le compilateur synthétise davantage de méthodes lorsque vous déclarez des enregistrements positionnels :

- Constructeur principal dont les paramètres correspondent aux paramètres positionnels de la déclaration d'enregistrement.
- Propriétés public init-only pour chaque paramètre d'un constructeur principal.
- `Deconstruct` Méthode permettant d'extraire des propriétés de l'enregistrement.

Données de température de build

Les données et les statistiques font partie des scénarios dans lesquels vous souhaiterez utiliser des enregistrements. Pour ce didacticiel, vous allez créer une application qui calcule le *degré de jours* pour différentes utilisations. Les *jours de degré* sont une mesure de chaleur (ou de l'absence de chaleur) sur une période de jours, de semaines ou de mois. Degré de suivi et prédiction de l'utilisation de l'énergie. Plus de plus chaud jours signifie plus de climatisation, et plus de jours plus froids signifient une utilisation plus grande du four. Degree Days aident à gérer les populations d'usines. Les jours sont corrélés à la croissance de la plante au fur et à mesure que les saisons évoluent. Le degree Days aide à effectuer le suivi des migrations animales pour les espèces qui voyagent pour correspondre au climat.

La formule est basée sur la température moyenne d'un jour donné et sur une température de ligne de base. Pour calculer le degré des jours dans le temps, vous avez besoin des températures haute et basse chaque jour pendant une période donnée. Commençons par créer une nouvelle application. Créez une nouvelle application console. Créez un nouveau type d'enregistrement dans un nouveau fichier nommé « DailyTemperature.cs » :

```
public record DailyTemperature(double HighTemp, double LowTemp);
```

Le code précédent définit un *enregistrement positionnel*. Vous avez créé un type référence qui contient deux propriétés : `HighTemp` , et `LowTemp` . Ces propriétés sont des *Propriétés init uniquement*, ce qui signifie qu'elles peuvent être définies dans le constructeur ou à l'aide d'un initialiseur de propriété. Le `DailyTemperature` type a également un *constructeur principal* qui a deux paramètres qui correspondent aux deux propriétés. Vous utilisez le constructeur principal pour initialiser un `DailyTemperature` enregistrement :

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

Vous pouvez ajouter vos propres propriétés ou méthodes à des enregistrements, y compris des enregistrements positionnels. Vous devez calculer la température moyenne pour chaque jour. Vous pouvez ajouter cette propriété à l' `DailyTemperature` enregistrement :

```
public record DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Vérifions que vous pouvez utiliser ces données. Ajoutez le code suivant à votre méthode `Main` :

```
foreach (var item in data)
    Console.WriteLine(item);
```

Exécutez votre application. vous verrez une sortie similaire à l'affichage suivant (plusieurs lignes supprimées pour l'espace) :

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }


DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

Le code précédent montre la sortie de la substitution de `ToString` synthétisé par le compilateur. Si vous préférez du texte différent, vous pouvez écrire votre propre version de `ToString` . Cela empêche le compilateur de synthétiser une version pour vous.

Degré de calcul en jours

Pour calculer les jours, vous prenez la différence par rapport à une température de référence et la température moyenne d'un jour donné. Pour mesurer la chaleur dans le temps, vous supprimez tous les jours où la température moyenne est inférieure à la ligne de base. Pour mesurer le froid au fil du temps, vous devez ignorer les jours où la température moyenne est supérieure à la ligne de base. Par exemple, les États-Unis utilisent 65F comme base pour les jours de chauffage et de refroidissement. Il s'agit de la température à laquelle aucun chauffage ou refroidissement n'est nécessaire. Si un jour a une température moyenne de 70F, ce jour est de 5 jours de refroidissement et 0 jour de chauffage. À l'inverse, si la température moyenne est 55F, ce jour correspond à 10 jours de chauffage et 0 jour de refroidissement.

Vous pouvez exprimer ces formules sous la forme d'une petite hiérarchie de types d'enregistrements : un type de jour abstrait et deux types concrets pour le chauffage des jours et le degré de refroidissement. Ces types peuvent également être des enregistrements positionnels. Ils prennent une température de référence et une séquence d'enregistrements de température quotidiens comme arguments pour le constructeur principal :

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);

public record HeatingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s => BaseTemperature -
s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s => s.Mean -
BaseTemperature);
}
```

L' `DegreeDays` enregistrement abstrait est la classe de base partagée pour les `HeatingDegreeDays` `CoolingDegreeDays` enregistrements et. Les déclarations de constructeur principales sur les enregistrements dérivés montrent comment gérer l'initialisation des enregistrements de base. Votre enregistrement dérivé déclare des paramètres pour tous les paramètres dans le constructeur principal d'enregistrement de base. L'enregistrement de base déclare et initialise ces propriétés. L'enregistrement dérivé ne les masque pas, mais crée et initialise uniquement les propriétés des paramètres qui ne sont pas déclarés dans son enregistrement de base. Dans cet exemple, les enregistrements dérivés n'ajoutent pas de nouveaux paramètres de constructeur principal. Testez votre

code en ajoutant le code suivant à votre `Main` méthode :

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

Vous obtenez une sortie similaire à l'affichage suivant :

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Définir des méthodes synthétisées par le compilateur

Votre code calcule le nombre correct de jours de chauffage et de refroidissement pendant cette période. Mais cet exemple montre pourquoi vous souhaiterez peut-être remplacer certaines des méthodes synthétisées pour les enregistrements. Vous pouvez déclarer votre propre version de l'une des méthodes synthétisées par le compilateur dans un type d'enregistrement, à l'exception de la méthode Clone. La méthode Clone a un nom généré par le compilateur et vous ne pouvez pas fournir une implémentation différente. Ces méthodes synthétisées incluent un constructeur de copie, les membres de l' System.IEquatable<T> interface, les tests d'égalité et d'inégalité, et GetHashCode() . À cet effet, vous allez synthétiser `PrintMembers` . Vous pouvez également déclarer votre propre `ToString` , mais `PrintMembers` offre une meilleure option pour les scénarios d'héritage. Pour fournir votre propre version d'une méthode synthétisée, la signature doit correspondre à la méthode synthétisée.

L' `TempRecords` élément dans la sortie de la console n'est pas utile. Elle affiche le type, mais rien d'autre. Vous pouvez modifier ce comportement en fournissant votre propre implémentation de la méthode synthétisée `PrintMembers` . La signature dépend des modificateurs appliqués à la `record` déclaration :

- Si un type d'enregistrement est `sealed` , la signature est `private bool PrintMembers(StringBuilder builder);`
- Si un type d'enregistrement n'est pas `sealed` et dérive de `object` (c'est-à-dire qu'il ne déclare pas d'enregistrement de base), la signature est `protected virtual bool PrintMembers(StringBuilder builder);`
- Si un type d'enregistrement n'est pas `sealed` et qu'il dérive d'un autre enregistrement, la signature est `protected override bool PrintMembers(StringBuilder builder);`

Ces règles sont plus faciles à comprendre pour comprendre l'objectif de `PrintMembers` . `PrintMembers` ajoute à une chaîne des informations sur chaque propriété d'un type d'enregistrement. Le contrat requiert des enregistrements de base pour ajouter leurs membres à l'affichage et suppose que les membres dérivés ajouteront leurs membres. Chaque type d'enregistrement synthétise un `ToString` remplacement qui ressemble à l'exemple suivant pour `HeatingDegreeDays` :

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Vous déclarez une `PrintMembers` méthode dans l' `DegreeDays` enregistrement qui n'imprime pas le type de la

collection :

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

La signature déclare une `virtual protected` méthode pour qu'elle corresponde à la version du compilateur. Ne vous inquiétez pas si vous recevez les accesseurs erronés ; le langage applique la signature correcte. Si vous oubliez les modificateurs corrects pour toute méthode synthétisée, le compilateur émet des avertissements ou des erreurs qui vous aident à obtenir la signature appropriée.

Mutation non destructrice

Les membres synthétisés dans un enregistrement positionnel ne modifient pas l'état de l'enregistrement. L'objectif est que vous pouvez créer plus facilement des enregistrements immuables. Examinez à nouveau les déclarations précédentes pour `HeatingDegreeDays` et `CoolingDegreeDays` . Les membres ajoutés effectuent des calculs sur les valeurs de l'enregistrement, mais ne changent pas d'État. Les enregistrements positionnels facilitent la création de types référence immuables.

La création de types référence immuables signifie que vous souhaiterez utiliser une mutation non destructrice. Vous créez des instances d'enregistrement similaires aux instances d'enregistrement existantes à l'aide d' `with` expressions. Ces expressions sont une construction de copie avec des affectations supplémentaires qui modifient la copie. Le résultat est une nouvelle instance d'enregistrement dans laquelle chaque propriété a été copiée à partir de l'enregistrement existant et éventuellement modifiée. L'enregistrement d'origine est inchangé.

Nous allons ajouter quelques fonctionnalités à votre programme qui illustrent des `with` expressions. Tout d'abord, nous allons créer un nouvel enregistrement pour calculer le nombre de jours croissants à l'aide des mêmes données. Le nombre de *jours croissants* utilise généralement 41F comme ligne de base et mesure les températures au-dessus de la ligne de base. Pour utiliser les mêmes données, vous pouvez créer un nouvel enregistrement similaire à `coolingDegreeDays` , mais avec une température de base différente :

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Vous pouvez comparer le nombre de degrés calculés aux nombres générés à l'aide d'une température de référence supérieure. N'oubliez pas que les enregistrements sont des *types référence* et que ces copies sont des copies superficielles. Le tableau des données n'est pas copié, mais les deux enregistrements font référence aux mêmes données. Ce fait est un avantage dans un autre scénario. Pour des jours de plus en bout, il est utile d'effectuer le suivi du total des 5 jours précédents. Vous pouvez créer des enregistrements avec différentes données sources à l'aide d' `with` expressions. Le code suivant génère une collection de ces accumulations, puis affiche les valeurs :

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Vous pouvez également utiliser `with` des expressions pour créer des copies d'enregistrements. Ne spécifiez aucune propriété entre les accolades de l' `with` expression. Cela signifie créer une copie et ne pas modifier les propriétés :

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Exécutez l'application finie pour afficher les résultats.

Résumé

Ce didacticiel a présenté plusieurs aspects des enregistrements. Les enregistrements fournissent une syntaxe concise pour les types de référence où l'utilisation fondamentale est le stockage des données. Pour les classes orientées objet, l'utilisation fondamentale est la définition des responsabilités. Ce didacticiel se concentre sur les *enregistrements positionnels*, où vous pouvez utiliser une syntaxe concise pour déclarer les propriétés d'initialisation uniquement pour un enregistrement. Le compilateur synthétise plusieurs membres de l'enregistrement pour copier et comparer des enregistrements. Vous pouvez ajouter tous les autres membres dont vous avez besoin pour vos types d'enregistrements. Vous pouvez créer des types d'enregistrements immuables sachant qu'aucun des membres générés par le compilateur ne fera passer l'État. Pour les enregistrements positionnels, les expressions facilitent la `with` prise en charge de la mutation non destructrice.

Les enregistrements ajoutent une autre façon de définir des types. `class` Les définitions vous permettent de créer des hiérarchies orientées objet qui se concentrent sur les responsabilités et le comportement des objets. Vous créez `struct` des types pour les structures de données qui stockent les données et sont suffisamment petits pour être copiés efficacement. Vous créez des enregistrements lorsque vous souhaitez une comparaison et une égalité basée sur les valeurs, que vous ne voulez pas copier de valeurs et que vous souhaitez utiliser des variables de référence.

Vous pouvez apprendre la description complète des enregistrements en lisant la spécification de type d'enregistrement proposé.

Didacticiel : Explorez des idées à l'aide d'instructions de niveau supérieur pour générer du code à mesure que vous en Apprenez

16/12/2020 • 12 minutes to read • Edit Online

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Découvrez les règles qui régissent l'utilisation des instructions de niveau supérieur.
- Utilisez les instructions de niveau supérieur pour explorer les algorithmes.
- Refactorisez les explorations dans des composants réutilisables.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 5, qui comprend le compilateur C# 9. Le compilateur C# 9 est disponible à partir de Visual Studio 2019 version 16,9 Preview 1 ou .net 5,0 SDK.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET Core.

Commencez à explorer

Les instructions de niveau supérieur vous permettent d'éviter la cérémonie supplémentaire requise en plaçant le point d'entrée de votre programme dans une méthode statique dans une classe. Le point de départ classique d'une nouvelle application console ressemble au code suivant :

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Le code précédent est le résultat de l'exécution de la `dotnet new console` commande et de la création d'une nouvelle application console. Ces 11 lignes contiennent une seule ligne de code exécutable. Vous pouvez simplifier ce programme avec la nouvelle fonctionnalité d'instructions de niveau supérieur. Cela vous permet de supprimer toutes les lignes, sauf deux, dans ce programme :

```
using System;

Console.WriteLine("Hello World!");
```

Cette action simplifie ce qui est nécessaire pour commencer à explorer de nouvelles idées. Vous pouvez utiliser des instructions de niveau supérieur pour les scénarios de script ou pour explorer. Une fois les principes de base opérationnels, vous pouvez commencer à refactoriser le code et créer des méthodes, des classes ou d'autres assemblys pour des composants réutilisables que vous avez générés. Les instructions de niveau supérieur

permettent d'effectuer des expérimentations rapides et des didacticiels débutants. Ils fournissent également un chemin lisse entre l'expérimentation et les programmes complets.

Les instructions de niveau supérieur sont exécutées dans l'ordre dans lequel elles apparaissent dans le fichier. Les instructions de niveau supérieur ne peuvent être utilisées que dans un seul fichier source dans votre application. Le compilateur génère une erreur si vous les utilisez dans plusieurs fichiers.

Créer un ordinateur de réponse Magic .NET

Pour ce didacticiel, nous allons créer une application console qui répond à une question « oui » ou « non » avec une réponse aléatoire. Vous allez créer les fonctionnalités pas à pas. Vous pouvez vous concentrer sur votre tâche plutôt que sur la cérémonie nécessaire à la structure d'un programme standard. Ensuite, une fois que vous êtes satisfait de la fonctionnalité, vous pouvez refactoriser l'application comme vous le souhaitez.

Un bon point de départ consiste à réécrire la question sur la console. Vous pouvez commencer par écrire le code suivant :

```
using System;

Console.WriteLine(args);
```

Vous ne déclarez pas de `args` variable. Pour le fichier source unique qui contient vos instructions de niveau supérieur, le compilateur reconnaît `args` les arguments de ligne de commande. Le type des arguments est un `string[]` , comme dans tous les programmes C#.

Vous pouvez tester votre code en exécutant la `dotnet run` commande suivante :

```
dotnet run -- Should I use top level statements in all my programs?
```

Les arguments qui suivent le `--` sur la ligne de commande sont passés au programme. Vous pouvez voir le type de la `args` variable, car il s'agit de l'impression sur la console :

```
System.String[]
```

Pour écrire la question dans la console, vous devez énumérer les arguments et les séparer par un espace. Remplacez l' `WriteLine` appel par le code suivant :

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Désormais, lorsque vous exécutez le programme, la question est correctement affichée sous la forme d'une chaîne d'arguments.

Répondre avec une réponse aléatoire

Après avoir renvoyé la question, vous pouvez ajouter le code pour générer la réponse aléatoire. Commencez par ajouter un tableau de réponses possibles :

```
string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};
```

Ce tableau a 12 réponses positives, six qui ne sont pas valides et six négatives. Ensuite, ajoutez le code suivant pour générer et afficher une réponse aléatoire à partir du tableau :

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Vous pouvez réexécuter l’application pour voir les résultats. Vous devriez voir une sortie semblable à la suivante :

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

Ce code répond aux questions, mais nous allons ajouter une fonctionnalité supplémentaire. Vous aimeriez que votre application de question simule une réflexion sur la réponse. Vous pouvez le faire en ajoutant un peu d’animations ASCII et en interrompant le travail. Ajoutez le code suivant après la ligne qui renvoie la question :

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

Vous devez également ajouter une `using` instruction en haut du fichier source :

```
using System.Threading.Tasks;
```

Les `using` instructions doivent être antérieures à toute autre instruction dans le fichier. Dans le cas contraire, il s’agit d’une erreur du compilateur. Vous pouvez réexécuter le programme et voir l’animation. Cela offre une meilleure expérience. Expérimentez la longueur du délai pour vous adapter à votre goût.

Le code précédent crée un ensemble de lignes en rotation séparées par un espace. L’ajout du `await` mot clé

indique au compilateur de générer le point d'entrée du programme en tant que méthode avec le `async` modificateur et retourne un System.Threading.Tasks.Task . Ce programme ne retourne pas de valeur, donc le point d'entrée du programme retourne un `Task` . Si votre programme retourne une valeur entière, vous devez ajouter une instruction return à la fin de vos instructions de niveau supérieur. Cette instruction return spécifie la valeur entière à retourner. Si vos instructions de niveau supérieur incluent une `await` expression, le type de retour devient System.Threading.Tasks.Task<TResult> .

Refactorisation à l'avenir

Votre programme doit ressembler au code suivant :

```
using System;
using System.Threading.Tasks;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Le code précédent est raisonnable. Il fonctionne. Mais il n'est pas réutilisable. Maintenant que l'application fonctionne, il est temps de tirer parti des parties réutilisables.

Un candidat est le code qui affiche l'animation en attente. Cet extrait de code peut devenir une méthode :

Vous pouvez commencer par créer une fonction locale dans votre fichier. Remplacez l'animation actuelle par le code suivant :

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

Le code précédent crée une fonction locale à l'intérieur de votre méthode main. Cela n'est toujours pas réutilisable. Ainsi, extrayez ce code dans une classe. Créez un nouveau fichier nommé *Utilities.cs* et ajoutez le code suivant :

```
using System;
using System.Threading.Tasks;

namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

Les instructions de niveau supérieur ne peuvent figurer que dans un seul fichier, et ce fichier ne peut pas contenir d'espaces de noms ou de types.

Enfin, vous pouvez nettoyer le code de l'animation pour supprimer des doublons :

```
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

Vous disposez maintenant d'une application complète et vous avez refactorisé les parties réutilisables pour une utilisation ultérieure.

Résumé

Les instructions de niveau supérieur facilitent la création de programmes simples à utiliser pour explorer les nouveaux algorithmes. Vous pouvez expérimenter des algorithmes en essayant différents extraits de code. Une fois que vous avez appris ce qui fonctionne, vous pouvez Refactoriser le code pour qu'il soit plus facile à gérer.

Les instructions de niveau supérieur simplifient les programmes basés sur les applications console. Il s'agit notamment des fonctions Azure, des actions GitHub et d'autres petits utilitaires.

Utilisez des critères spéciaux pour générer le comportement de votre classe pour un meilleur code

16/12/2020 • 18 minutes to read • Edit Online

Les fonctionnalités de critères spéciaux en C# fournissent une syntaxe pour exprimer vos algorithmes. Vous pouvez utiliser ces techniques pour implémenter le comportement dans vos classes. Vous pouvez combiner la conception de classes orientée objet avec une implémentation orientée données pour fournir un code concis lors de la modélisation d'objets réels.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Exprimez vos classes orientées objet à l'aide de modèles de données.
- Implémentez ces modèles à l'aide des fonctionnalités de critères spéciaux de C#.
- Tirez parti des diagnostics du compilateur pour valider votre implémentation.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 5, y compris le compilateur C# 9,0. Le compilateur C# 9,0 est disponible à partir de Visual Studio 2019 version 16,8 Preview ou de la version préliminaire du kit de développement logiciel (SDK) .net 5,0.

Créer une simulation d'un canal de verrouillage

Dans ce didacticiel, vous allez générer une classe C# qui simule un verrou de canal. Brièvement, un canal de verrouillage est un appareil qui soulève et abaisse les canots lorsqu'ils se déplacent entre deux étirés d'eau à différents niveaux. Un verrou a deux portes et un mécanisme pour modifier le niveau d'eau.

Dans son fonctionnement normal, un bateau entre l'un des portails, tandis que le niveau d'eau du verrou correspond au niveau de l'eau sur le côté du bateau. Une fois dans le verrou, le niveau d'eau est modifié pour correspondre au niveau d'eau où le bateau laisse le verrou. Une fois que le niveau d'eau correspond à ce côté, la porte du côté de sortie s'ouvre. Les mesures de sécurité assurent qu'un opérateur ne peut pas créer une situation dangereuse dans le canal. Le niveau d'eau ne peut être modifié que lorsque les deux portes sont fermées. Au plus une porte peut être ouverte. Pour ouvrir une porte, le niveau d'eau dans le verrou doit correspondre au niveau d'eau en dehors de la porte qui est ouverte.

Vous pouvez générer une classe C# pour modéliser ce comportement. Une `CanalLock` classe prend en charge les commandes pour ouvrir ou fermer l'une ou l'autre des portes. Il aurait d'autres commandes pour augmenter ou diminuer l'eau. La classe doit également prendre en charge les propriétés pour lire l'état actuel des portes et du niveau d'eau. Vos méthodes implémentent les mesures de sécurité.

Définir une classe

Vous allez générer une application console pour tester votre `CanalLock` classe. Créez un nouveau projet de console pour .NET 5 à l'aide de Visual Studio ou de l'interface de commande .NET. Ensuite, ajoutez une nouvelle classe et nommez-la `CanalLock` . Ensuite, concevez votre API publique, mais laissez les méthodes non implémentées :

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } = WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}
```

Le code précédent Initialise l'objet afin que les deux portes soient fermées et que le niveau d'eau soit faible. Ensuite, écrivez le code de test suivant dans votre `Main` méthode pour vous guider lors de la création d'une première implémentation de la classe :

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");
Console.WriteLine(canalGate);

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Ensuite, ajoutez une première implémentation de chaque méthode dans la `CanalLock` classe. Le code suivant implémente les méthodes de la classe sans se préoccuper des règles de sécurité. Vous ajouterez des tests de sécurité plus tard :

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

Les tests que vous avez écrits jusqu'à présent passent. Vous avez implémenté les concepts de base. À présent, écrivez un test pour la première condition d'échec. À la fin des tests précédents, les deux portes sont fermées et le niveau d'eau est défini sur faible. Ajoutez un test pour essayer d'ouvrir la porte supérieure :

```
Console.WriteLine("==============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Ce test échoue car la porte s'ouvre. En guise de première implémentation, vous pouvez résoudre le problème à l'aide du code suivant :

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the water is low");
}
```

Vos tests réussissent. Toutefois, à mesure que vous ajoutez d'autres tests, vous ajouterez de plus en plus de `if` clauses et testerez différentes propriétés. Bientôt, ces méthodes seront trop compliquées à mesure que vous ajouterez des conditions supplémentaires.

Implémenter les commandes avec des modèles

Une meilleure méthode consiste à utiliser des *modèles* pour déterminer si l'objet est dans un état valide pour exécuter une commande. Vous pouvez exprimer si une commande est autorisée en tant que fonction de trois variables : l'état de la porte, le niveau de l'eau et le nouveau paramètre :

NOUVEAU PARAMÈTRE	ÉTAT DE LA PORTE	NIVEAU D'EAU	RÉSULTATS
Fermé	Fermé	Importante	Fermé
Fermé	Fermé	Faible	Fermé
Fermé	Ouvrir	Importante	Ouvrir
~~Fermé~~	~~Ouvrir~~	~~Faible~~	~~Fermé~~
Ouvrir	Fermé	Importante	Ouvrir
Ouvrir	Fermé	Faible	Fermé (erreur)
Ouvrir	Ouvrir	Importante	Ouvrir
~~Ouvrir~~	~~Ouvrir~~	~~Faible~~	~~Fermé (erreur)~~

Les quatrième et dernière lignes de la table ont du texte barré, car elles ne sont pas valides. Le code que vous ajoutez maintenant doit s'assurer que la porte de haute-mer n'est jamais ouverte lorsque l'eau est faible. Ces États

peuvent être codés comme une expression de commutateur unique (n'oubliez pas que `false` indique « fermé ») :

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the water
is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Essayez cette version. Vos tests réussissent, en validant le code. Le tableau complet indique les combinaisons possibles d'entrées et de résultats. Cela signifie que vous et d'autres développeurs pouvez consulter rapidement le tableau et voir que vous avez couvert toutes les entrées possibles. Encore plus facile, le compilateur peut également vous aider. Après avoir ajouté le code précédent, vous pouvez voir que le compilateur génère un avertissement : *CS8524* indique que l'expression de switch ne couvre pas toutes les entrées possibles. La raison de cet avertissement est que l'une des entrées est un `enum` type. Le compilateur interprète « toutes les entrées possibles » comme toutes les entrées du type sous-jacent, généralement `int` . Cette `switch` expression vérifie uniquement les valeurs déclarées dans le `enum` . Pour supprimer l'avertissement, vous pouvez ajouter un modèle de rejet tous pour le dernier ARM de l'expression. Cette condition lève une exception, car elle indique une entrée non valide :

```
_  => throw new InvalidOperationException("Invalid internal state"),
```

Le bras du commutateur précédent doit être le dernier dans votre `switch` expression, car il correspond à toutes les entrées. Essayez en la déplaçant plus tôt dans l'ordre. Cela provoque une erreur du compilateur *CS8510* pour le code inaccessible dans un modèle. La structure naturelle des expressions de commutateur permet au compilateur de générer des erreurs et des avertissements pour les erreurs éventuelles. Le « filet de sécurité » du compilateur facilite la création d'un code correct en moins d'itérations, et la liberté de combiner des bras de commutateur avec des caractères génériques. Le compilateur émettra des erreurs si votre combinaison produit des bras inaccessibles et des avertissements si vous supprimez un ARM qui est nécessaire.

La première modification consiste à combiner tous les bras où la commande doit fermer la porte ; Cette autorisation est toujours autorisée. Ajoutez le code suivant en tant que premier bras dans votre expression de commutateur :

```
(false, _, _) => false,
```

Une fois que vous avez ajouté le bras du commutateur précédent, vous obtenez quatre erreurs de compilation, une sur chacun des bras où la commande est `false` . Ces bras sont déjà couverts par le bras qui vient d'être ajouté. Vous pouvez supprimer ces quatre lignes en toute sécurité. Vous avez souhaité que ce nouveau bras de commutateur remplace ces conditions.

Ensuite, vous pouvez simplifier les quatre bras où la commande doit ouvrir la porte. Dans les deux cas où le niveau d'eau est élevé, la porte peut être ouverte. (Dans un, il est déjà ouvert.) Un cas où le niveau d'eau est faible lève une exception et l'autre ne doit pas se produire. Il doit être possible de lever la même exception en toute sécurité si le verrouillage de l'eau est déjà dans un État non valide. Vous pouvez effectuer les simplifications suivantes pour ces bras :

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the water is
low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Exécutez à nouveau vos tests, et ils réussissent. Voici la version finale de la `SetHighGate` méthode :

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _,    _)               => false,
        (true, _,     WaterLevel.High) => true,
        (true, false, WaterLevel.Low)  => throw new InvalidOperationException("Cannot open high gate when the
water is low"),
        _                              => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Implémentez vous-même des modèles

Maintenant que vous avez vu la technique, renseignez les `SetLowGate` `SetWaterLevel` méthodes et vous-même. Commencez par ajouter le code suivant pour tester les opérations non valides sur ces méthodes :

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Réexécutez votre application. Vous pouvez voir que les nouveaux tests échouent et que le canal de verrouillage est dans un État non valide. Essayez d'implémenter vous-même les méthodes restantes. La méthode permettant de définir la porte inférieure doit être semblable à la méthode pour définir la porte supérieure. La méthode qui modifie le niveau d'eau a des contrôles différents, mais elle doit suivre une structure similaire. Il peut s'avérer utile d'utiliser le même processus pour la méthode qui définit le niveau d'eau. Commencez avec les quatre entrées : l'état des deux portes, l'état actuel du niveau d'eau et le nouveau niveau d'eau demandé. L'expression de Switch doit commencer par :

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

Vous aurez 16 bras de commutateur total à remplir. Ensuite, testez et simplifiez.

Avez-vous fait des méthodes comme ceci ?

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new InvalidOperationException("Cannot open high gate when the
water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new InvalidOperationException("Cannot lower
water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new InvalidOperationException("Cannot raise
water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Vos tests doivent réussir et le canal de verrouillage doit fonctionner en toute sécurité.

Résumé

Dans ce didacticiel, vous avez appris à utiliser des critères spéciaux pour vérifier l'état interne d'un objet avant d'appliquer les modifications apportées à cet État. Vous pouvez vérifier les combinaisons de propriétés. Une fois que vous avez créé des tables pour l'une de ces transitions, vous testez votre code, puis vous simplifiez la lisibilité et la maintenabilité. Ces refactorisations initiales peuvent suggérer des refactorisations supplémentaires qui valident l'état interne ou gèrent d'autres modifications de l'API. Ce didacticiel combine des classes et des objets avec une approche plus orientée données et basée sur les modèles pour implémenter ces classes.

Utiliser l'interpolation de chaîne pour construire des chaînes mises en forme

06/05/2020 • 15 minutes to read • Edit Online

Ce tutoriel explique comment utiliser l'interpolation de chaîne en C# pour insérer des valeurs dans une chaîne de résultat unique. Vous allez écrire un code en C# et afficher les résultats de la compilation et de l'exécution du code. Le tutoriel contient une série de leçons qui expliquent comment insérer des valeurs dans une chaîne et mettre en forme ces valeurs de différentes façons.

Ce tutoriel suppose que vous disposez d'un ordinateur que vous pouvez utiliser pour le développement. Le didacticiel .NET Hello World en 10 minutes contient des instructions pour la configuration de votre environnement de développement local sur Windows, Linux ou MacOS. Vous pouvez également compléter la version interactive de ce didacticiel dans votre navigateur.

Créer une chaîne interpolée

Créez un répertoire nommé *interpolated*. Faites-en le répertoire actif et exécutez la commande suivante à partir d'une fenêtre de console :

```
dotnet new console
```

Cette commande crée une nouvelle application console .NET Core dans le répertoire actuel.

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez la ligne `Console.WriteLine("Hello World!");` par le code qui suit, en remplaçant `<name>` par votre nom :

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Essayez ce code en tapant `dotnet run` dans la fenêtre de console. Quand vous exécutez le programme, celui-ci affiche une chaîne unique qui inclut votre nom dans le message d'accueil. La chaîne qui se trouve dans l'appel de méthode WriteLine est une *expression de chaîne interpolée*. C'est un genre de modèle qui vous permet de construire une chaîne unique (appelée *chaîne de résultat*) à partir d'une chaîne qui comprend du code incorporé. Les chaînes interpolées sont particulièrement utiles pour insérer des valeurs dans une chaîne ou pour concaténer (joindre) des chaînes.

Cet exemple simple contient les deux éléments que chaque chaîne interpolée doit avoir :

- Un littéral de chaîne qui commence par le caractère `$` avant ses guillemets ouvrants. Il ne peut pas y avoir d'espace entre le symbole `$` et les guillemets. (Si vous voulez voir ce qui se passe si vous en incluez un, insérez un espace après le caractère `$`, enregistrez le fichier et réexécutez le programme en tapant `dotnet run` dans la fenêtre de console. Le compilateur C# affiche un message d'erreur « Erreur CS1056 : caractère inattendu '$' ».)
- Une ou plusieurs *expressions d'interpolation*. Une expression d'interpolation est indiquée par des accolades ouvrantes et fermantes (`{` et `}`). Vous pouvez placer n'importe quelle expression C# qui retourne une valeur (notamment `null`) à l'intérieur des accolades.

Essayons quelques autres exemples d'interpolation de chaîne avec d'autres types de données.

Inclure différents types de données

Dans la section précédente, vous avez utilisé l'interpolation de chaîne pour insérer une chaîne à l'intérieur d'une autre. Le résultat d'une expression d'interpolation peut toutefois être de n'importe quel type de données. Nous allons insérer des valeurs de différents types de données dans une chaîne interpolée.

Dans l'exemple suivant, nous commençons par définir une classe comme type de données `Vegetable` avec la `Name` propriété et la `ToString` méthode, qui remplace le comportement de la méthode Object.ToString(). Le `public` modificateur d'accès rend cette méthode disponible pour tout code client pour obtenir la représentation sous forme `Vegetable` de chaîne d'une instance. Dans l'exemple, `Vegetable.ToString` la méthode retourne la valeur de `Name` la propriété qui est initialisée dans `Vegetable` le constructeur:

```
public Vegetable(string name) => Name = name;
```

Ensuite, nous créons une instance de `Vegetable` la classe `item` nommée à l'aide de l' `new` opérateur et fournissons un `Vegetable` nom pour le constructeur :

```
var item = new Vegetable("eggplant");
```

Enfin, nous incluons la variable `item` dans une chaîne interpolée qui contient également une valeur DateTime, une valeur Decimal et une valeur d' énumération `Unit`. Remplacez tout le code C# dans votre éditeur par le code suivant, puis utilisez la commande `dotnet run` pour l'exécuter :

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
   }
}
```

Notez que l'expression d'interpolation `item` dans la chaîne interpolée résout le texte « eggplant » (aubergine) dans la chaîne de résultat. En effet, lorsque le résultat de l'expression n'est pas de type chaîne, il est résolu en une chaîne de la façon suivante :

- Si l'expression d'interpolation donne une valeur `null`, une chaîne vide («» ou String.Empty) est utilisée.
- Si l'expression d'interpolation ne donne pas une valeur `null`, en général, la méthode `ToString` du type de résultat est appelée. Vous pouvez tester cela en mettant à jour l'implémentation de la méthode `Vegetable.ToString`. Il n'est peut-être même pas nécessaire d'implémenter la méthode `ToString` étant

donné que chaque type a déjà une implémentation de cette méthode. Pour tester cela, commentez la définition de la méthode `Vegetable.ToString` dans l'exemple (en la faisant précéder d'un symbole de commentaire `//`). Dans la sortie, la chaîne « eggplant » (aubergines) est remplacée par le nom de type complet (« Vegetable » dans cet exemple), ce qui est le comportement par défaut de la méthode Object.ToString(). Le comportement par défaut de la méthode `ToString` pour une valeur d'énumération est de retourner la représentation sous forme de chaîne de la valeur.

Dans la sortie de cet exemple, la date est trop précise (le prix des aubergines ne change pas chaque seconde) et la valeur du prix n'indique pas la devise locale. Dans la section suivante, vous découvrirez comment résoudre ces problèmes en contrôlant le format des représentations sous forme de chaînes des résultats des expressions.

Contrôler la mise en forme des expressions d'interpolation

Dans la section précédente, deux chaînes à la mise en forme incorrecte ont été insérées dans la chaîne de résultat. L'une était une valeur de date et d'heure pour laquelle seule la date était appropriée. La deuxième était un prix qui n'indiquait pas la devise locale. Ces deux problèmes sont faciles à résoudre. En utilisant l'interpolation de chaîne, vous pouvez spécifier des *chaînes de format* qui contrôlent la mise en forme de types particuliers. Modifiez l'appel à `Console.WriteLine` de l'exemple précédent de façon à inclure les chaînes de format pour les expressions de date et de prix, comme indiqué dans la ligne de code suivante :

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

Vous spécifiez une chaîne de format en plaçant après l'expression d'interpolation un signe deux-points (« : ») et la chaîne de format. « d » est une chaîne de format de date et d'heure standard qui représente le format de date courte. « C2 » est une chaîne de format numérique standard qui représente un nombre sous la forme d'une valeur monétaire avec deux chiffres après la virgule décimale.

Plusieurs types dans les bibliothèques .NET prennent en charge un ensemble prédéfini de chaînes de format. Il s'agit notamment de tous les types numériques et des types de date et d'heure. Pour obtenir une liste complète des types qui prennent en charge les chaînes de format, consultez Chaînes de format et types de bibliothèque de classes .NET dans l'article Mise en forme des types dans .NET.

Essayez de modifier les chaînes de format dans votre éditeur de texte et, à chaque modification, relancez le programme pour voir comment celles-ci affectent la mise en forme de la date et de l'heure et la valeur numérique. Remplacez le « d » dans `{date:d}` par « t » (pour afficher le format d'heure courte), « y » (pour afficher l'année et mois) et « yyyy » (pour afficher l'année sous forme de nombre à quatre chiffres). Remplacez le « C2 » dans `{price:C2}` par « e » (pour la notation exponentielle) et « F3 » (pour une valeur numérique avec trois chiffres après la virgule).

En plus de contrôler la mise en forme, vous pouvez contrôler la largeur de champ et l'alignement des chaînes mises en forme qui sont incluses dans la chaîne de résultat. Dans la section suivante, vous allez découvrir comment effectuer cette opération.

Contrôler la largeur de champ et l'alignement des expressions d'interpolation

En règle générale, lorsque le résultat d'une expression d'interpolation est mise en forme en chaîne, cette chaîne est inclue dans une chaîne de résultats sans espace de début ou de fin. Contrôler la largeur d'un champ et l'alignement du texte vous permet de rendre une sortie plus lisible, en particulier quand vous utilisez un jeu de données. Pour voir cela, remplacez tout le code dans votre éditeur de texte par le code suivant, puis tapez `dotnet run` pour exécuter le programme :

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

Les noms des auteurs sont alignés à gauche, et leurs titres sont alignés à droite. Vous spécifiez l'alignement en ajoutant une virgule (« , ») après une expression d'interpolation et en spécifiant la largeur de champ *minimum*. Si la valeur spécifiée est un nombre positif, le champ est aligné à droite. Si c'est un nombre négatif, le champ est aligné à gauche.

Essayez de supprimer les signes négatifs du code `{"Author",-25}` et `{title.Key,-25}`, puis réexécutez l'exemple, comme avec le code suivant :

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

Cette fois, les informations sur l'auteur sont alignées à droite.

Vous pouvez combiner un spécificateur d'alignement et une chaîne de format pour une même expression d'interpolation. Pour cela, spécifiez d'abord l'alignement, puis un signe deux-points et la chaîne de format. Remplacez tout le code à l'intérieur de la méthode `Main` par le code suivant, qui affiche trois chaînes mises en forme avec des largeurs de champ définies. Ensuite, exécutez le programme en entrant la commande `dotnet run`.

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

La sortie ressemble à ceci :

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

Vous avez terminé le tutoriel sur l'interpolation de chaîne.

Pour plus d'informations, consultez la rubrique Interpolation de chaîne et le tutoriel Interpolation de chaîne en C#.

Interpolation de chaîne en C#

18/03/2020 • 10 minutes to read • Edit Online

Ce tutoriel vous montre comment utiliser une interpolation de chaîne pour mettre en forme et inclure des résultats d'expressions dans une chaîne de résultat. Les exemples supposent que vous êtes familiarisé avec les concepts de base de C# et la mise en forme des types .NET. Si vous ne connaissez pas l'interpolation de chaînes ou la mise en forme de types .NET, consultez d'abord le tutoriel interactif sur l'interpolation de chaînes. Pour plus d'informations sur la mise en forme des types dans .NET, consultez la rubrique Mise en forme des types dans .NET.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Introduction

La fonctionnalité Interpolation de chaîne s'appuie sur la fonctionnalité Mise en forme composite et fournit une syntaxe plus lisible et plus pratique pour inclure des résultats d'expressions mises en forme dans une chaîne de résultat.

Pour identifier un littéral de chaîne comme chaîne interpolée, préfixez-la du symbole `$`. Vous pouvez incorporer n'importe quelle expression C# valide qui retourne une valeur dans une chaîne interpolée. Dans l'exemple suivant, dès qu'une expression est évaluée, son résultat est converti en chaîne et est inclus dans une chaîne de résultat :

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is
{CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Comme le montre l'exemple, vous incluez une expression dans une chaîne interpolée en la plaçant entre des accolades :

```
{<interpolationExpression>}
```

Les chaînes interpolées prennent en charge toutes les fonctions de la fonctionnalité Mise en forme de chaîne composite. Elles constituent ainsi une alternative plus lisible à l'utilisation de la méthode String.Format.

Comment spécifier une chaîne de format pour une expression d'interpolation

Vous spécifiez une chaîne de format prise en charge par le type du résultat d'expression en plaçant un signe deux-

points (« : ») et la chaîne de format après l'expression d'interpolation :

```
{<interpolationExpression>:<formatString>}
```

L'exemple suivant montre comment spécifier des chaînes de format standard et personnalisées pour des expressions qui produisent des résultats de type date/heure ou numérique :

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

Pour plus d'informations, consultez la section Composant de chaîne de format de la rubrique Mise en forme composite. Cette section fournit des liens vers les rubriques qui décrivent les chaînes de format standard et personnalisées prises en charge par les types de base .NET.

Comment contrôler la largeur de champ et l'alignement de l'expression d'interpolation mise en forme

Vous spécifiez la largeur minimale du champ et l'alignement du résultat de l'expression mise en forme en plaçant une virgule (« , ») et l'expression constante après l'expression d'interpolation :

```
{<interpolationExpression>,<alignment>}
```

Si la valeur de *l'alignement* est positive, le résultat de l'expression mise en forme est aligné à droite ; si la valeur est négative, il est aligné à gauche.

Si vous devez spécifier à la fois un alignement et une chaîne de format, commencez par le composant d'alignement :

```
{<interpolationExpression>,<alignment>:<formatString>}
```

L'exemple suivant montre comment spécifier l'alignement ; il utilise des caractères de barre verticale (« | ») pour délimiter les champs texte :

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

Comme le montre l'exemple de sortie, si la longueur du résultat d'expression mise en forme dépasse la largeur de

champ spécifiée, la valeur de *l'alignement* est ignorée.

Pour plus d'informations, consultez la section Composant d'alignement de la rubrique Mise en forme composite.

Comment utiliser des séquences d'échappement dans une chaîne interpolée

Les chaînes interpolées prennent en charge toutes les séquences d'échappement qui peuvent être utilisés dans les littéraux de chaîne ordinaires. Pour plus d'informations, consultez Séquences d'échappement de chaîne.

Pour interpréter les séquences d'échappement littéralement, utilisez un littéral de chaîne textuelle. Une chaîne textuelle interpolée `$` commence par `@` le personnage suivi du personnage. En commençant par C 8.0, `$` `@` vous pouvez utiliser les `$@"..."` jetons et les jetons dans n'importe quel ordre : les deux et `@$"..."` sont valides interpolés chaînes textuelles.

Pour inclure une accolade, « { » ou «} », dans une chaîne de résultat, utilisez deux accolades, « {{ » ou «}} ». Pour plus d'informations, consultez la section Échappement des accolades de la rubrique Mise en forme composite.

L'exemple suivant montre comment inclure des accolades dans une chaîne de résultat et construire une chaîne interpolée textuelle :

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}}
sets.");

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

Comment utiliser un opérateur conditionnel ternaire `?:` dans une expression d'interpolation

Comme le caractère deux-points (« : ») a une signification spéciale dans un élément avec une expression d'interpolation, pour utiliser un opérateur conditionnel dans une expression, placez-la entre parenthèses, comme le montre l'exemple suivant :

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

Comment créer une chaîne de résultat spécifique à une culture avec une interpolation de chaîne

Par défaut, une chaîne interpolée utilise la culture active définie par la propriété CultureInfo.CurrentCulture pour toutes les opérations de mise en forme. Utilisez la conversion implicite d'une chaîne interpolée en une instance de System.FormattableString et appelez sa méthode ToString(IFormatProvider) pour créer une chaîne de résultat

spécifique à une culture. L'exemple suivant montre comment effectuer cette opération :

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US       5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL        17-05-18 15:44:55      31.415.926,536
//            05/17/2018 15:44:55      31,415,926.536
```

Comme le montre l'exemple, vous pouvez utiliser une même instance de FormattableString pour générer plusieurs chaînes de résultat pour différentes cultures.

Comment créer une chaîne de résultat avec la culture invariante

Avec la méthode FormattableString.ToString(IFormatProvider), vous pouvez utiliser la méthode statique FormattableString.Invariant pour résoudre une chaîne interpolée en une chaîne de résultat pour InvariantCulture. L'exemple suivant montre comment effectuer cette opération :

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusion

Ce tutoriel décrit des scénarios courants d'utilisation de l'interpolation de chaîne. Pour plus d'informations sur l'interpolation de chaîne, consultez la rubrique Interpolation de chaîne. Pour plus d'informations sur la mise en forme des types dans .NET, consultez les rubriques Mise en forme des types dans .NET et Mise en forme composite.

Voir aussi

- String.Format
- System.FormattableString
- System.IFormattable
- Cordes

Didacticiel : mettre à jour les interfaces avec les méthodes d'interface par défaut en C# 8,0

02/11/2020 • 11 minutes to read • Edit Online

Depuis C# 8.0 sur .NET Core 3.0, vous pouvez définir une implémentation lorsque vous déclarez un membre d'une interface. Le scénario le plus courant consiste à ajouter de manière sécurisée des membres à une interface déjà publiée et utilisée par de nombreux clients.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Étendre des interfaces de manière sécurisée en ajoutant des méthodes avec des implémentations
- Créer des implémentations paramétrables pour fournir une plus grande flexibilité
- Permettre aux implémenteurs de fournir une implémentation plus spécifique sous la forme d'un remplacement

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET Core, y compris le compilateur C# 8,0. Le compilateur C# 8,0 est disponible à partir de Visual Studio 2019 version 16,3 ou du Kit de développement logiciel (SDK) .net Core 3,0.

Présentation du scénario

Ce tutoriel commence par la version 1 d'une bibliothèque de relation client. Vous pouvez obtenir l'application de démarrage à partir de notre dépôt samples sur GitHub. L'entreprise qui a créé cette bibliothèque espérait la voir adoptée par les clients ayant des applications existantes. Elle a fourni des définitions d'interface minimales destinées à être implémentées par les utilisateurs de sa bibliothèque. Voici la définition d'interface pour un client :

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

Elle a défini une deuxième interface qui représente une commande :

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

À partir de ces interfaces, l'équipe a pu générer une bibliothèque pour les utilisateurs permettant de créer une meilleure expérience pour ses clients. Son objectif était de créer une relation plus étroite avec les clients existants et d'améliorer ses relations avec les nouveaux clients.

À présent, il est temps de mettre à niveau la bibliothèque pour la prochaine version. L'une des fonctionnalités demandées permet une remise de fidélité pour les clients qui ont un grand nombre de commandes. Cette nouvelle

remise de fidélité est appliquée chaque fois qu'un client passe une commande. La remise spécifique est une propriété de chaque client. Chaque implémentation de `ICustomer` peut définir des règles différentes pour la remise de fidélité.

La façon la plus naturelle d'ajouter cette fonctionnalité consiste à améliorer l'interface `ICustomer` avec une méthode pour appliquer une remise de fidélité. Cette suggestion de conception a posé problème aux développeurs expérimentés : «les interfaces sont immuables une fois qu'elles ont été publiées ! Il s'agit d'une modification avec rupture ! » C# 8.0 ajoute des *implémentations d'interface par défaut* pour la mise à niveau des interfaces. Les auteurs de bibliothèque peuvent ajouter de nouveaux membres à l'interface et fournir une implémentation par défaut pour ces membres.

Les implémentations d'interface par défaut permettent aux développeurs de mettre à niveau une interface tout en laissant la possibilité aux implémenteurs de substituer cette implémentation. Les utilisateurs de la bibliothèque peuvent accepter l'implémentation par défaut en tant que modification sans rupture. Si leurs règles métier sont différentes, ils peuvent opérer un remplacement.

Mettre à niveau avec les méthodes d'interface par défaut

L'équipe s'est mise d'accord sur l'implémentation par défaut la plus vraisemblable : une remise de fidélité pour les clients.

La mise à niveau doit fournir la fonctionnalité permettant de définir deux propriétés : le nombre de commandes nécessaires pour pouvoir bénéficier de la remise et le pourcentage de celle-ci. Cela en fait un scénario parfait pour les méthodes d'interface par défaut. Vous pouvez ajouter une méthode à l' `ICustomer` interface et fournir l'implémentation la plus probable. Toutes les implémentations existantes et nouvelles peuvent utiliser l'implémentation par défaut ou fournir la leur.

Tout d'abord, ajoutez la nouvelle méthode à l'interface, y compris le corps de la méthode :

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

L'auteur de la bibliothèque a écrit un premier test pour vérifier l'implémentation :

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Notez la partie suivante du test :

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Ce cast de `SampleCustomer` en `ICustomer` est nécessaire. La classe `SampleCustomer` n'a pas besoin de fournir une implémentation pour `ComputeLoyaltyDiscount` ; elle est fournie par l'interface `ICustomer`. Toutefois, la classe `SampleCustomer` n'hérite pas les membres de ses interfaces. Cette règle n'a pas changé. Pour qu'il soit possible d'appeler n'importe quelle méthode déclarée et implémentée dans l'interface, la variable doit être du même type que l'interface, `ICustomer` dans cet exemple.

Fournir le paramétrage

C'est un bon début. Toutefois, l'implémentation par défaut est trop restrictive. De nombreux consommateurs de ce système peuvent choisir des seuils différents pour le nombre d'achats, une durée différente pour la qualité de membre ou un pourcentage de remise différent. Vous pouvez procurer une meilleure expérience de mise à niveau à davantage de clients en fournissant un moyen de définir ces paramètres. Nous allons ajouter une méthode statique qui définit ces trois paramètres de contrôle de l'implémentation par défaut :

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

De nombreuses nouvelles fonctionnalités de langage sont présentées dans ce petit fragment de code. Les interfaces peuvent maintenant inclure des membres statiques, notamment des champs et des méthodes. Différents modificateurs d'accès sont également activés. Les champs supplémentaires sont privés, tandis que la nouvelle méthode est publique. Tout modificateur est autorisé sur les membres d'interface.

Les applications qui utilisent la formule générale pour le calcul de la remise de fidélité, mais des paramètres différents, n'ont pas besoin de fournir une implémentation personnalisée ; elles peuvent définir les arguments par le biais d'une méthode statique. Par exemple, le code suivant définit une « évaluation du client » qui récompense tout client membre depuis plus d'un mois :

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Étendre l'implémentation par défaut

Le code que vous avez ajouté jusqu'à présent a fourni une implémentation pratique pour les scénarios où les utilisateurs veulent quelque chose comme l'implémentation par défaut ou pour fournir un ensemble de règles non lié. En guise de dernière fonctionnalité, nous allons refactoriser le code un peu pour permettre les scénarios où les utilisateurs sont susceptibles de vouloir générer l'implémentation par défaut.

Imaginez une start-up qui souhaite attirer de nouveaux clients. Elle offre une remise de 50 % sur la première commande d'un nouveau client. Pour leur part, les clients existants bénéficient de la remise standard. L'auteur de la bibliothèque doit déplacer l'implémentation par défaut vers une méthode `protected static` afin que toute classe qui implémente cette interface puisse réutiliser le code dans son implémentation. L'implémentation par défaut du membre d'interface appelle également cette méthode partagée :

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Dans une implémentation d'une classe qui implémente cette interface, le remplacement peut appeler la méthode d'assistance statique et étendre cette logique pour fournir la remise « nouveau client » :

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Vous pouvez voir l'intégralité du code terminé dans notre dépôt d'exemples sur GitHub. Vous pouvez obtenir l'application de démarrage à partir de notre dépôt samples sur GitHub.

Ces nouvelles fonctionnalités signifient que les interfaces peuvent être mises à jour de manière sécurisée quand il existe une implémentation par défaut raisonnable pour les nouveaux membres. Concevez soigneusement les interfaces pour exprimer des idées fonctionnelles uniques pouvant être implémentées par plusieurs classes. Cela facilite la mise à niveau de ces définitions d'interface quand de nouvelles exigences sont découvertes pour cette même idée fonctionnelle.

Didacticiel : combiner les fonctionnalités dans lors de la création de classes à l'aide d'interfaces avec les méthodes d'interface par défaut

14/05/2020 • 16 minutes to read • Edit Online

Depuis C# 8.0 sur .NET Core 3.0, vous pouvez définir une implémentation lorsque vous déclarez un membre d'une interface. Cette fonctionnalité offre de nouvelles fonctionnalités qui vous permettent de définir des implémentations par défaut pour les fonctionnalités déclarées dans les interfaces. Les classes peuvent choisir quand remplacer les fonctionnalités, quand utiliser les fonctionnalités par défaut et quand ne pas déclarer la prise en charge des fonctionnalités discrètes.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Créez des interfaces avec des implémentations qui décrivent des fonctionnalités discrètes.
- Créez des classes qui utilisent les implémentations par défaut.
- Créez des classes qui remplacent une partie ou la totalité des implémentations par défaut.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET Core, y compris le compilateur C# 8,0. Le compilateur C# 8,0 est disponible à partir de Visual Studio 2019 version 16,3ou du Kit de développement logiciel (SDK) .net Core 3,0 ou version ultérieure.

Limitations des méthodes d'extension

L'une des façons dont vous pouvez implémenter le comportement qui s'affiche dans le cadre d'une interface est de définir des méthodes d'extension qui fournissent le comportement par défaut. Les interfaces déclarent un ensemble minimal de membres tout en fournissant une plus grande surface d'exposition pour toute classe qui implémente cette interface. Par exemple, les méthodes d'extension dans Enumerable fournissent l'implémentation pour toute séquence comme source d'une requête LINQ.

Les méthodes d'extension sont résolues au moment de la compilation, à l'aide du type déclaré de la variable. Les classes qui implémentent l'interface peuvent fournir une meilleure implémentation pour toute méthode d'extension. Les déclarations de variables doivent correspondre au type d'implémentation pour permettre au compilateur de choisir cette implémentation. Lorsque le type au moment de la compilation correspond à l'interface, les appels de méthode sont résolus à la méthode d'extension. Une autre préoccupation avec les méthodes d'extension est que ces méthodes sont accessibles partout où la classe qui contient les méthodes d'extension est accessible. Les classes ne peuvent pas déclarer si elles doivent ou ne doivent pas fournir des fonctionnalités déclarées dans les méthodes d'extension.

À compter de C# 8,0, vous pouvez déclarer les implémentations par défaut en tant que méthodes d'interface. Ensuite, chaque classe utilise automatiquement l'implémentation par défaut. Toute classe pouvant fournir une meilleure implémentation peut remplacer la définition de méthode d'interface par un meilleur algorithme. Dans un sens, cette technique ressemble à la façon dont vous pouvez utiliser les méthodes d'extension.

Dans cet article, vous allez découvrir comment les implémentations d'interface par défaut permettent de nouveaux scénarios.

Concevoir l'application

Prenons l'exemple d'une application domotique. Vous disposez probablement de nombreux types d'éclairages et d'indicateurs qui peuvent être utilisés dans l'ensemble de la maison. Chaque lumière doit prendre en charge les API pour les activer et les désactiver, et pour signaler l'état actuel. Certains éclairages et indicateurs peuvent prendre en charge d'autres fonctionnalités, telles que :

- Activez le voyant, puis désactivez-le après une minuterie.
- Faire clignoter la lumière pendant une période donnée.

Certaines de ces fonctionnalités étendues peuvent être émulées sur les appareils qui prennent en charge l'ensemble minimal. Qui indique la fourniture d'une implémentation par défaut. Pour les appareils qui ont plus de fonctionnalités intégrées, le logiciel de l'appareil utilise les fonctionnalités natives. Pour les autres lumières, elles peuvent choisir d'implémenter l'interface et d'utiliser l'implémentation par défaut.

Les membres d'interface par défaut sont une meilleure solution pour ce scénario que les méthodes d'extension. Les auteurs de classe peuvent contrôler les interfaces qu'ils choisissent d'implémenter. Les interfaces qu'ils choisissent sont disponibles en tant que méthodes. En outre, étant donné que les méthodes d'interface par défaut sont virtuelles par défaut, la distribution de méthode choisit toujours l'implémentation dans la classe.

Créons le code pour illustrer ces différences.

Créer des interfaces

Commencez par créer l'interface qui définit le comportement de toutes les lumières :

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Un dispositif d'éclairage de surcharge de base peut implémenter cette interface comme indiqué dans le code suivant :

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Dans ce didacticiel, le code ne pilote pas les appareils IoT, mais émule ces activités en écrivant des messages sur la console. Vous pouvez explorer le code sans automatiser votre maison.

Ensuite, nous allons définir l'interface pour une lumière qui peut s'éteindre automatiquement après un délai d'attente :

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Vous pouvez ajouter une implémentation de base à la lumière de la surcharge, mais une meilleure solution consiste à modifier cette définition d'interface pour fournir une `virtual` implémentation par défaut :

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

En ajoutant cette modification, la `OverheadLight` classe peut implémenter la fonction timer en déclarant la prise en charge de l'interface :

```
public class OverheadLight : ITimerLight { }
```

Un type de lumière différent peut prendre en charge un protocole plus sophistiqué. Il peut fournir sa propre implémentation pour `TurnOnFor` , comme illustré dans le code suivant :

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

Contrairement aux méthodes de classe virtuelle de substitution, la déclaration de `TurnOnFor` dans la `HalogenLight` classe n'utilise pas le `override` mot clé.

Fonctionnalités Mix et match

Les avantages des méthodes d'interface par défaut deviennent plus clairs lorsque vous introduisez des fonctionnalités plus avancées. L'utilisation d'interfaces vous permet de mélanger et de faire correspondre des fonctionnalités. Il permet également à chaque auteur de classe de choisir entre l'implémentation par défaut et une implémentation personnalisée. Nous allons ajouter une interface avec une implémentation par défaut pour une lumière clignotante :

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for IBlinkingLight.Blink.");
    }
}
```

L'implémentation par défaut permet à tout éclairage de clignoter. La lumière de la surcharge peut ajouter des fonctionnalités de minuterie et de clignotement à l'aide de l'implémentation par défaut :

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Un nouveau type de lumière, le `LEDLight` prend directement en charge la fonction de minuteur et la fonction Blink. Ce style clair implémente à la fois les `ITimerLight` interfaces et et `IBlinkingLight` substitue la `Blink` méthode :

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink funtion.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Un `ExtraFancyLight` peut prendre en charge les fonctions de clignotement et de minuteur directement :

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Le `HalogenLight` que vous avez créé précédemment ne prend pas en charge le clignotement. Par conséquent, n'ajoutez pas `IBlinkingLight` à la liste des interfaces prises en charge.

Détecter les types de lumière à l'aide de critères spéciaux

Nous allons ensuite écrire du code de test. Vous pouvez utiliser la fonctionnalité de mise en correspondance des modèles de C# pour déterminer les capacités d'un éclairage en examinant les interfaces qu'il prend en charge. La méthode suivante exerce les fonctionnalités prises en charge de chaque lumière :

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

Le code suivant dans votre `Main` méthode crée chaque type de lumière dans la séquence et teste ce qui est clair :

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Comment le compilateur détermine la meilleure implémentation

Ce scénario montre une interface de base sans implémentation. L'ajout d'une méthode dans l' `ILight` interface introduit de nouvelles complexités. Les règles de langage régissant les méthodes d'interface par défaut réduisent l'effet sur les classes concrètes qui implémentent plusieurs interfaces dérivées. Nous allons améliorer l'interface d'origine avec une nouvelle méthode pour montrer comment cela modifie son utilisation. Chaque témoin lumineux peut signaler son état d'alimentation comme une valeur énumérée :

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

L'implémentation par défaut suppose qu'il n'y ait pas de puissance :

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Ces modifications sont compilées correctement, même si le `ExtraFancyLight` déclare la prise en charge de l' `ILight` interface et des deux interfaces dérivées, `ITimerLight` et `IBlinkingLight` . Il n'y a qu'une seule implémentation « la plus proche » déclarée dans l' `ILight` interface. Toute classe qui a déclaré une substitution devient l'implémentation « la plus proche ». Vous avez vu des exemples dans les classes précédentes qui a remplacé les membres d'autres interfaces dérivées.

Évitez de substituer la même méthode dans plusieurs interfaces dérivées. Si vous procédez ainsi, un appel de

méthode ambigu est créé chaque fois qu'une classe implémente les deux interfaces dérivées. Le compilateur ne peut pas choisir une méthode unique pour qu'il génère une erreur. Par exemple, si `IBlinkingLight` et implémentent tous deux la `ITimerLight` substitution de `PowerStatus` , le doit `OverheadLight` fournir une substitution plus spécifique. Sinon, le compilateur ne peut pas choisir entre les implémentations dans les deux interfaces dérivées. Vous pouvez généralement éviter cette situation en conservant les définitions d'interface de petite taille et en vous concentrant sur une seule fonctionnalité. Dans ce scénario, chaque fonctionnalité d'une lumière est sa propre interface ; plusieurs interfaces sont héritées uniquement par les classes.

Cet exemple illustre un scénario dans lequel vous pouvez définir des fonctionnalités discrètes qui peuvent être mélangées dans des classes. Vous déclarez tout ensemble de fonctionnalités prises en charge en déclarant les interfaces prises en charge par une classe. L'utilisation de méthodes d'interface par défaut virtuelles permet aux classes d'utiliser ou de définir une implémentation différente pour toute ou partie des méthodes d'interface. Cette fonctionnalité de langage offre de nouvelles façons de modéliser les systèmes réels que vous créez. Les méthodes d'interface par défaut offrent un moyen plus clair d'exprimer des classes connexes qui peuvent combiner et faire correspondre des fonctionnalités différentes à l'aide d'implémentations virtuelles de ces fonctionnalités.

Index et plages

16/12/2020 • 11 minutes to read • Edit Online

Les plages et les index fournissent une syntaxe concise pour accéder à des éléments ou des plages uniques dans une séquence.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Utiliser la syntaxe pour les plages dans une séquence.
- Comprendre les décisions de conception pour le début et la fin de chaque séquence.
- Découvrir des scénarios pour les types Index et Range.

Prise en charge linguistique pour les index et les plages

Cette prise en charge de langage s'appuie sur deux nouveaux types et deux nouveaux opérateurs :

- System.Index représente un index au sein d'une séquence.
- Index de l'opérateur End `^` , qui spécifie qu'un index est relatif à la fin d'une séquence.
- System.Range représente une sous-plage d'une séquence.
- Opérateur de plage `..` , qui spécifie le début et la fin d'une plage comme opérandes.

Commençons par les règles concernant les indices. Prenons pour exemple un tableau `sequence`. L'index `0` est identique à l'index `sequence[0]`. L'index `^0` est identique à l'index `sequence[sequence.Length]`. L'expression `sequence[^0]` lève une exception, tout comme le `sequence[sequence.Length]` fait. Pour n'importe quel nombre `n`, l'index `^n` est identique à l'index `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Vous pouvez récupérer le dernier mot avec l'index `^1`. Ajoutez le code suivant sous l'initialisation :

```
Console.WriteLine($"The last word is {words[^1]}");
```

Une plage spécifie son *début* et sa *fin*. Les plages sont exclusives, ce qui signifie que la *fin* n'est pas incluse dans la plage. La plage `[0..^0]` représente la plage dans son intégralité, tout comme `[0..sequence.Length]` représente la plage entière.

Le code suivant crée une sous-plage qui comporte les mots « quick », « brown » et « fox » et va de `words[1]` à `words[3]`. L'élément `words[4]` n'est pas dans la plage. Ajoutez le code suivant à la même méthode. Copiez-le et collez-le en bas de la fenêtre interactive.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Le code suivant retourne la plage avec « Lazy » et « Dog ». et comprend `words[^2]` et `words[^1]`. L'index de fin `words[^0]` n'est pas inclus. Ajoutez également le code suivant :

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Les exemples suivants créent des plages ouvertes au début, à la fin ou les deux :

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the, "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Vous pouvez également déclarer des plages ou index comme variables. La variable peut ensuite être utilisée à l'intérieur des caractères `[` et `]` :

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

L'exemple suivant montre un grand nombre des raisons de ces choix. Modifiez `x`, `y` et `z` pour essayer différentes combinaisons. Quand vous effectuez des essais, utilisez des valeurs de telle sorte que `x` soit inférieur à `y` et `y` inférieur à `z` pour avoir des combinaisons valides. Ajoutez le code suivant à une nouvelle méthode. Essayez différentes combinaisons :

```
int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through
{y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");
```

Prise en charge des types d'index et de plages

Les index et les plages fournissent une syntaxe claire et concise permettant d'accéder à un élément unique ou à une plage d'éléments dans une séquence. Une expression d'index retourne généralement le type des éléments d'une séquence. Une expression de plage retourne généralement le même type de séquence que la séquence source.

Tout type qui fournit un indexeur avec un Index Range paramètre ou prend explicitement en charge les index ou les plages respectivement. Un indexeur qui accepte un Range paramètre unique peut retourner un type de séquence différent, tel que System.Span<T> .

> **IMPORTANT**
>
> Les performances du code à l'aide de l'opérateur de plage dépendent du type de l'opérande de séquence.
>
> La complexité du temps de l'opérateur de plage dépend du type de séquence. Par exemple, si la séquence est un `string` tableau ou, le résultat est une copie de la section spécifiée de l'entrée, de sorte que la complexité de l'heure est *O (n)* (où n est la longueur de la plage). En revanche, s'il s'agit d'un System.Span<T> ou d'un System.Memory<T> , le résultat fait référence au même magasin de stockage, ce qui signifie qu'il n'y a pas de copie et que l'opération est *O (1)*.
>
> En plus de la complexité du temps, cela provoque des allocations et des copies supplémentaires, ce qui a un impact sur les performances. Dans le code dépendant des performances, envisagez d'utiliser `Span<T>` ou `Memory<T>` comme type de séquence, puisque l'opérateur de plage ne les alloue pas.

Un type est **compté** s'il a une propriété nommée `Length` ou `Count` avec un accesseur Get accessible et un type de retour `int` . Un type pouvant être compté qui ne prend pas explicitement en charge les index ou les plages peut fournir une prise en charge implicite pour eux. Pour plus d'informations, consultez les sections prise en charge d' index implicite et prise en charge de plage implicite de la proposition. Les plages utilisant la prise en charge de plage implicite retournent le même type de séquence que la séquence source.

Par exemple, les types .NET suivants prennent en charge les index et les plages : String , Span<T> et ReadOnlySpan<T> . Le List<T> prend en charge les index mais ne prend pas en charge les plages.

Array a un comportement plus nuance. Les tableaux à une seule dimension prennent en charge les index et les plages. Les tableaux multidimensionnels ne le sont pas. L'indexeur d'un tableau multidimensionnel possède plusieurs paramètres, et non un seul paramètre. Les tableaux en escalier, également connus sous le terme de tableau de tableaux, prennent en charge les plages et les indexeurs. L'exemple suivant montre comment itérer une sous-section rectangulaire d'un tableau en escalier. Il itère la section au centre, en excluant les trois dernières lignes, et les deux dernières colonnes de chaque ligne sélectionnée :

```
var jagged = new int[10][]
{
   new int[10] { 0,  1, 2, 3, 4, 5, 6, 7, 8, 9},
   new int[10] { 10,11,12,13,14,15,16,17,18,19},
   new int[10] { 20,21,22,23,24,25,26,27,28,29},
   new int[10] { 30,31,32,33,34,35,36,37,38,39},
   new int[10] { 40,41,42,43,44,45,46,47,48,49},
   new int[10] { 50,51,52,53,54,55,56,57,58,59},
   new int[10] { 60,61,62,63,64,65,66,67,68,69},
   new int[10] { 70,71,72,73,74,75,76,77,78,79},
   new int[10] { 80,81,82,83,84,85,86,87,88,89},
   new int[10] { 90,91,92,93,94,95,96,97,98,99},
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Dans tous les cas, l'opérateur de plage pour Array alloue un tableau pour stocker les éléments retournés.

Scénarios pour les index et les plages

Vous utiliserez souvent des plages et des index lorsque vous souhaitez analyser une partie d'une plus grande séquence. La nouvelle syntaxe est plus claire pour la lecture exacte de la partie de la séquence. La fonction locale `MovingAverage` prend un Range comme argument. La méthode énumère ensuite simplement cette plage lors du calcul des valeurs minimale, maximale et moyenne. Essayez le code suivant dans votre projet :

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();
```

Tutoriel : Exprimer plus clairement une intention de conception avec les types référence Nullable et non Nullable

02/11/2020 • 19 minutes to read • Edit Online

C# 8,0 introduit les types de référence Nullable, qui complètent les types de référence de la même façon que les types valeur Nullable complètent les types valeur. Pour déclarer une variable comme étant un **type référence Nullable**, on ajoute `?` au type. Par exemple, `string?` représente une `string` Nullable. Vous pouvez utiliser ces nouveaux types pour exprimer plus clairement votre intention de conception : certaines variables *doivent toujours avoir une valeur*, d'autres *peuvent ne pas en avoir*.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- incorporer des types référence Nullable et non Nullable dans vos conceptions ;
- activer les contrôles de type référence Nullable dans l'ensemble de votre code ;
- écrire du code permettant au compilateur d'appliquer ces décisions de conception ;
- utiliser la fonctionnalité de référence Nullable dans vos propres conceptions.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET Core, y compris le compilateur C# 8,0. Le compilateur C# 8,0 est disponible avec Visual Studio 2019ou .net Core 3,0.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET Core.

Incorporer des types référence Nullable dans des conceptions

Dans ce tutoriel, vous allez générer une bibliothèque qui modélise la réalisation d'une enquête. Le code utilise des types référence Nullable et non Nullable pour représenter les concepts du monde réel. Les questions de l'enquête ne peuvent jamais être Null. Si la personne interrogée ne souhaite pas répondre à une question, Les réponses peuvent être `null` dans ce cas.

Le code que vous allez écrire pour cet exemple exprime cette intention, que le compilateur se charge d'appliquer.

Créer l'application et activer les types référence Nullable

Créez une application console dans Visual Studio ou en ligne de commande avec `dotnet new console`. Nommez l'application `NullableIntroduction`. Une fois que vous avez créé l'application, vous devez spécifier que l'intégralité du projet compile dans un contexte d' **annotation Nullable**activé. Ouvrez le fichier *. csproj* et ajoutez un `Nullable` élément à l' `PropertyGroup` élément. Affectez-lui la valeur `enable`. Vous devez opter pour la fonctionnalité de **types référence Nullable** , même dans les projets C# 8,0. En effet, une fois la fonctionnalité activée, les déclarations de variables référence existantes deviennent des **types référence non Nullable**. Bien que cette décision permette de détecter les problèmes dans lesquels le code existant peut ne pas avoir de contrôles nuls corrects, il risque de ne pas refléter précisément votre intention de conception d'origine :

```
<Nullable>enable</Nullable>
```

Concevoir les types de l'application

Cette application d'enquête implique la création d'un certain nombre de classes :

- une classe qui modélise la liste des questions ;
- une classe qui modélise la liste des personnes contactées pour l'enquête ;
- une classe qui modélise les réponses d'une personne ayant répondu à l'enquête.

Ces types utilisent des types référence Nullable et non Nullable pour indiquer quels membres sont requis et lesquels sont facultatifs. Les types référence Nullable communiquent clairement cette intention de conception :

- Les questions qui font partie de l'enquête ne peuvent jamais être Null : poser une question vide n'aurait pas de sens.
- Les personnes interrogées ne peuvent jamais être Null. Vous souhaitez faire un suivi des personnes que vous avez contactées, même de celles qui ont refusé de participer.
- La réponse à une question peut être Null. Les personnes interrogées peuvent refuser de répondre à certaines questions ou à la totalité d'entre elles.

Si vous avez programmé en C#, vous pouvez être habitué à référencer des types qui autorisent des `null` valeurs que vous avez peut-être manqué d'autres possibilités pour déclarer des instances non Nullable :

- La collection de questions doit être non Nullable.
- La collection de personnes interrogées doit être non Nullable.

À mesure que vous écrivez le code, vous verrez qu'un type de référence qui n'accepte pas les valeurs NULL comme valeur par défaut pour les références évite les erreurs courantes qui pourraient mener à des NullReferenceException s. L'une des leçons de ce didacticiel est que vous avez pris des décisions concernant les variables qui pouvaient ou non être `null` . Avant, le langage ne fournissait pas de syntaxe permettant d'exprimer ces décisions ; maintenant, si.

L'application que vous allez générer effectue les étapes suivantes :

1. Crée une enquête et y ajoute des questions.
2. Crée un ensemble Pseudo-aléatoire de personnes interrogées pour l'enquête.
3. Contacte les répondants jusqu'à ce que la taille de l'enquête terminée atteigne le numéro d'objectif.
4. Écrit des statistiques importantes sur les réponses de l'enquête.

Générer l'enquête avec des types de référence Nullable et non Nullable

La première partie du code crée l'enquête. Vous allez écrire des classes pour modéliser une question et une enquête. L'enquête comporte trois types de questions, en fonction du format de la réponse : réponses par oui ou non, réponses chiffrées et réponses textuelles. Créez une `public SurveyQuestion` classe :

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

Le compilateur interprète chaque déclaration de variable de type référence comme un type de référence **non Nullable** pour le code dans un contexte d'annotation Nullable activé. Un premier avertissement apparaît lorsque l'on ajoute des propriétés pour le texte et le type de la question avec le code suivant :

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Comme `QuestionText` n'est pas initialisé, le compilateur émet l'avertissement selon lequel une propriété non Nullable n'a pas été initialisée. Or, la conception exige que le texte de la question soit non Null. Ajoutez un constructeur pour l'initialiser, ainsi que la valeur `QuestionType`. Une fois terminée, la définition de classe se présente ainsi :

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}
```

Le fait d'ajouter le constructeur supprime l'avertissement. L'argument du constructeur étant également un type référence non Nullable, le compilateur ne génère pas d'avertissements.

Ensuite, créez une classe `public` nommée `SurveyRun`. Cette classe contient une liste d'objets `SurveyQuestion` et de méthodes permettant d'ajouter des questions à l'enquête, comme l'illustre le code suivant :

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}
```

Comme tout à l'heure, il faut initialiser l'objet de liste sur une valeur non Null pour éviter que le compilateur

n'émette un avertissement. Il n'y a pas de contrôle de type Null dans la deuxième surcharge de `AddQuestion`, car ce n'est pas nécessaire : vous avez déclaré cette variable comme étant non Nullable. Sa valeur ne peut pas être `null`.

Basculez vers *Program.cs* dans votre éditeur et remplacez le contenu de `Main` par les lignes de code suivantes :

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that
happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Étant donné que le projet entier est dans un contexte d'annotation Nullable activé, vous obtenez des avertissements quand vous transmettez `null` à une méthode qui attend un type de référence non Nullable. Regardez le résultat en ajoutant la ligne suivante à `Main` :

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Créer des personnes interrogées et obtenir des réponses à l'enquête

Ensuite, écrivez le code qui génère des réponses à l'enquête. Ce processus implique plusieurs petites tâches :

1. Créer une méthode qui génère des objets personne interrogée, représentant les personnes à qui il a été demandé de répondre à l'enquête.
2. Créer une logique pour simuler les questions posées à une personne interrogée et les réponses obtenues, ou noter qu'une personne interrogée n'a pas répondu.
3. Répéter le processus jusqu'à ce que les personnes interrogées soient en nombre suffisant.

Il vous faut une classe qui représente les réponse à l'enquête. Ajoutez-la maintenant. Activez la prise en charge Nullable. Ajoutez une propriété `Id` et un constructeur qui l'initialise, comme dans le code suivant :

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Ensuite, ajoutez une méthode `static` pour créer de nouveaux participants en générant un ID aléatoire :

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

La responsabilité principale de cette classe consiste à générer les réponses d'un participant aux questions de l'enquête, ce qui se décompose en plusieurs étapes :

1. Demander à participer à l'enquête. Si la personne refuse, renvoyer une réponse manquante (Null).
2. Poser chaque question et enregistrer la réponse. Chacune des réponses peut également être manquante (Null).

Ajoutez le code suivant à la classe `SurveyResponse` :

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

Les réponses à l'enquête sont stockées dans un `Dictionary<int, string>?`, qui peut donc être Null. Vous utilisez la nouvelle fonctionnalité du langage pour déclarer votre intention de conception, à la fois au compilateur et à toute personne qui lira votre code. Si vous déréférencez jamais `surveyResponses` sans vérifier la `null` valeur en premier, vous obtenez un avertissement du compilateur. La méthode `AnswerSurvey` ne génère pas d'avertissement, car le compilateur peut déterminer que la variable `surveyResponses` a été définie avant sur une valeur non Null.

L'utilisation de `null` pour les réponses manquantes met en évidence un point important pour travailler avec les types référence nullable : votre objectif n'est pas de supprimer toutes les valeurs `null` à partir de votre programme. Au lieu de cela, votre objectif est de vous assurer que le code que vous écrivez exprime l'intention de votre conception. Les valeurs manquantes sont un concept nécessaire à exprimer dans votre code. La valeur `null` est une méthode évidente pour exprimer les valeurs manquantes. Essayer de supprimer toutes les valeurs `null` mène uniquement à la définition d'une autre façon d'exprimer ces valeurs manquantes sans `null`.

Ensuite, il reste à écrire la méthode `PerformSurvey` dans la classe `SurveyRun`. Ajoutez le code suivant à la classe

`SurveyRun` :

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Là encore, le choix d'une `List<SurveyResponse>?` Nullable indique que la réponse peut être Null, et que l'enquête n'a pour le moment été menée auprès de personne. Des personnes interrogées sont ajoutées jusqu'à ce qu'elles soient suffisamment nombreuses à avoir accepté.

La dernière étape consiste à ajouter un appel pour effectuer l'enquête à la fin de la méthode `Main` :

```
surveyRun.PerformSurvey(50);
```

Examiner les réponses à l'enquête

La dernière étape consiste à afficher les résultats de l'enquête, ce qui suppose d'ajouter du code à la plupart des classes déjà écrites. Ce code montre l'intérêt de distinguer les types référence Nullable et non Nullable. Commencez par ajouter les deux membres expression-bodied suivants à la classe `SurveyResponse` :

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";
```

Étant donné que `surveyResponses` est un type référence Nullable, les vérifications NULL sont nécessaires avant de le déréférencer. La `Answer` méthode retourne une chaîne qui n'accepte pas les valeurs NULL. nous devons donc aborder le cas d'une réponse manquante à l'aide de l'opérateur de fusion Null.

Ensuite, ajoutez ces trois membres expression-bodied à la classe `SurveyRun` :

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

Le membre `AllParticipants` doit prendre en compte le fait que la variable `respondents` peut être Null, mais pas la valeur de retour. Si vous modifiez cette expression en supprimant `??` et la séquence vide qui suit, le compilateur émet un avertissement, indiquant que la méthode peut retourner `null` et sa signature de retour un type non Nullable.

Enfin, ajoutez la boucle suivante en bas de la méthode `Main` :

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} : {answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

Aucun contrôle `null` n'est nécessaire dans ce code, car vous avez conçu les interfaces sous-jacentes de sorte qu'elles retournent toutes des types référence non Nullable.

Obtenir le code

Pour obtenir le code du tutoriel complet, consultez notre dépôt samples dans le dossier csharp/NullableIntroduction.

Faites des essais en modifiant les déclarations de type entre les types référence Nullable et non Nullable. Examinez les différents avertissements générés, qui visent à empêcher de déréférencer accidentellement un `null`.

Étapes suivantes

En savoir plus en migrant une application existante pour utiliser des types de référence nullable :

Mettre à niveau une application pour utiliser des types de référence nullable

Découvrez comment utiliser le type de référence Nullable quand vous utilisez Entity Framework :

> Notions de base de l'Entity Framework Core : utilisation des types de référence Nullable

Tutorial: Migrer le code existant avec des types de référence nuls

18/03/2020 • 26 minutes to read • Edit Online

C# 8 introduit les **types référence Nullable**, qui viennent compléter les types référence de la même façon que les types valeur Nullable complètent les types valeur. Pour déclarer une variable comme étant un **type référence Nullable**, on ajoute `?` au type. Par exemple, `string?` représente une `string` Nullable. Vous pouvez utiliser ces nouveaux types pour exprimer plus clairement votre intention de conception : certaines variables *doivent toujours avoir une valeur*, d'autres *peuvent ne pas en avoir*. Toutes les variables existantes d'un type de référence sont interprétées comme un type de référence non nullable.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Activer les vérifications de référence null lorsque vous travaillez avec du code.
- Diagnostiquer et corriger les différents avertissements liés aux valeurs null.
- Gérer l'interface entre les contextes compatibles avec nullable et non compatibles avec nullable.
- Contrôler les contextes d'annotation nullable.

Conditions préalables requises

Vous aurez besoin de configurer votre machine pour exécuter .NET Core, y compris le compilateur C 8.0. Le compilateur C 8 est disponible à partir de Visual Studio 2019 version 16.3 ou .NET Core 3.0 SDK.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET Core.

Explorer l'exemple d'application

L'exemple d'application que vous allez migrer est une application web de lecteur de flux RSS. Il lit à partir d'un flux RSS unique et affiche des résumés des articles les plus récents. Vous pouvez sélectionner l'un des articles pour visiter le site. L'application est relativement nouvelle, mais a été écrite avant que les types de référence nullable ne soient disponibles. Les décisions de conception de l'application constituent des principes solides, mais ne tirent pas parti de cette fonctionnalité de langage importante.

L'exemple d'application inclut une bibliothèque de tests unitaires qui valide les fonctionnalités principales de l'application. Ce projet facilite la mise à niveau en toute sécurité, si vous modifiez l'implémentation basée sur les avertissements générés. Vous pouvez télécharger l'exemple de démarrage à partir du référentiel GitHub dotnet/samples.

Votre objectif de migration d'un projet doit être de tirer parti des nouvelles fonctionnalités de langage afin d'exprimer clairement votre intention sur la possibilité de valeur null des variables et faire cela de manière à ce que le compilateur ne génère d'avertissements lorsque le contexte d'annotation nullable et le contexte d'avertissement nullable sont définis sur `enabled`.

Mettre à niveau les projets vers C# 8

La première étape consiste à déterminer l'étendue de la tâche de migration. Commencez par la mise à niveau du projet vers C# 8.0 (ou version ultérieure). Ajoutez `LangVersion` l'élément au Groupe de propriété dans les deux fichiers csproj pour le projet Web et le projet de test unitaire :

```
<LangVersion>8.0</LangVersion>
```

La mise à niveau de la version de langage sélectionne C# 8.0, mais n'active pas le contexte d'annotation nullable ou le contexte d'avertissement nullable. Régénérez le projet pour vous assurer qu'il se génère sans avertissement.

Une bonne étape suivante consiste à activer le contexte d'annotation nullable et de voir combien d'avertissements sont générés. Ajoutez l'élément suivant aux fichiers csproj dans la solution, directement sous l'élément `LangVersion` :

```
<Nullable>enable</Nullable>
```

Effectuez une build de test et notez la liste d'avertissements. Dans cette petite application, le compilateur génère cinq avertissements, il est donc probable que vous laissiez le contexte d'annotation nullable activé et commenciez à résoudre les avertissements pour l'ensemble du projet.

Cette stratégie fonctionne uniquement pour les projets plus petits. Pour tous les projets plus volumineux, le nombre d'avertissements générés par l'activation du contexte d'annotation nullable pour l'intégralité de la base de code rend plus difficile de résoudre les avertissements de manière systématique. Pour les grands projets d'entreprise, vous aurez souvent besoin de migrer un projet à la fois. Dans chaque projet, migrez une classe ou un fichier à la fois.

Les avertissements contribuent à découvrir l'intention de conception d'origine

Il existe deux classes qui génèrent plusieurs avertissements. Commencez avec la classe `NewsStoryViewModel`. Supprimez l'élément `Nullable` des deux fichiers csproj afin de pouvoir limiter l'étendue des avertissements aux sections de code sur lesquelles vous travaillez. Ouvrez le fichier *NewsStoryViewModel.cs*, puis ajoutez les directives suivantes pour activer le contexte d'annotation nullable pour `NewsStoryViewModel` et restaurez-le en suivant cette définition de classe :

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```

Ces deux directives vous aident à concentrer vos efforts de migration. Les avertissements nullables sont générés pour la zone de code sur laquelle vous travaillez activement. Vous les laissez activés jusqu'à ce que vous soyez prêt à activer les avertissements pour l'ensemble du projet. Vous devriez utiliser `restore` plutôt que la valeur `disable` afin de ne pas désactiver accidentellement le contexte ultérieurement lorsque de l'activation des annotations nullables pour la totalité du projet. Une fois que vous avez activé le contexte d'annotation nullable pour la totalité du projet, vous pouvez supprimer tous les pragmas `#nullable` à partir de ce projet.

La classe `NewsStoryViewModel` est un objet de transfert de données (DTO) et deux des propriétés sont des chaînes de lecture/écriture :

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

Ces deux propriétés provoquent `CS8618`, « Propriété non-nullable initialisée ». Cela est suffisamment clair : les deux propriétés `string` ont la valeur par défaut `null` lorsqu'un `NewsStoryViewModel` est construit. Il est important de découvrir comment les objets `NewsStoryViewModel` sont construits. En examinant cette classe, vous ne pouvez pas savoir si la valeur `null` fait partie de la conception, ou si ces objets sont définis sur des valeurs non null lors de leur création. Les articles de presse sont créés dans la méthode `GetNews` de la classe `NewsService` :

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

Il se passe beaucoup de choses dans le bloc de code précédent. Cette application utilise le package NuGet AutoMapper pour construire un élément d'article de presse à partir d'un `ISyndicationItem`. Vous avez découvert que les éléments d'articles de presse sont construits et que les propriétés sont définies dans cette seule instruction. Cela signifie que la conception de `NewsStoryViewModel` indique que ces propriétés ne doivent jamais avoir la valeur `null`. Ces propriétés doivent être des **types de référence non nullable**. C'est ce qui correspond le mieux à l'intention de conception d'origine. En fait, n'importe quel `NewsStoryViewModel` *est* correctement instancié avec des valeurs non null. Ce qui fait du code d'initialisation suivant un correctif valide :

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

L'attribution de `Title` et `Uri` à `default` qui est `null` pour le type `string` ne change pas le comportement d'exécution du programme. `NewsStoryViewModel` est toujours construit avec des valeurs null, mais maintenant le compilateur ne signale aucun avertissement. **L'opérateur indulgent en null**, le caractère `!` qui suit l'expression `default` indique au compilateur que l'expression précédente n'est pas null. Cette technique peut être opportune lorsque d'autres modifications forcent des modifications beaucoup plus importantes `NewsStoryViewModel` à une base de code, mais dans cette application il ya une solution relativement rapide et meilleure: Faire le type immuable où toutes les propriétés sont définies dans le constructeur. Dans `NewsStoryViewModel`, effectuez les changements suivants :

```
#nullable enable
    public class NewsStoryViewModel
    {
        public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
            (Published, Title, Uri) = (published, title, uri);

        public DateTimeOffset Published { get; }
        public string Title { get; }
        public string Uri { get; }
    }
#nullable restore
```

Une fois cette opération effectuée, vous devez mettre à jour le code qui configure AutoMapper afin qu'il utilise le

constructeur, plutôt que de définir des propriétés. Ouvrez `NewsService.cs` et recherchez le code suivant en bas du fichier :

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

Ce code mappe les propriétés de l'objet `ISyndicationItem` aux propriétés `NewsStoryViewModel`. Vous souhaitez qu'AutoMapper assure le mappage à l'aide d'un constructeur à la place. Remplacez le code ci-dessus par la configuration Automapper suivante :

```
#nullable enable
    public class NewsStoryProfile : Profile
    {
        public NewsStoryProfile()
        {
            // Create the AutoMapper mapping profile between the 2 objects.
            // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
            CreateMap<ISyndicationItem, NewsStoryViewModel>()
                .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
        }
```

Notez qu'étant donné que cette classe est petite, et que vous l'avez examinée avec soin, vous devez activer la directive `#nullable enable` au-dessus de cette déclaration de classe. La modification apportée au constructeur peut avoir rompu quelque chose, il est donc judicieux d'exécuter tous les tests et de tester l'application avant de continuer.

Le premier ensemble de modifications vous a montré comment effectuer la détection lorsque la conception d'origine indiquait que les variables ne devaient pas être définies sur `null`. Cette technique est appelée **correcte par construction**. Vous déclarez qu'un objet et ses propriétés ne peuvent pas être `null` quand il est construit. L'analyse du flux du compilateur fournit la garantie que ces propriétés ne sont pas définies sur `null` après la construction. Notez que ce constructeur est appelé par du code externe, que ce code **oublie la valeur nullable**. La nouvelle syntaxe ne fournit pas la vérification lors de l'exécution. Le code externe peut contourner l'analyse du flux du compilateur.

Parfois, la structure d'une classe fournit des différents indices sur l'intention. Ouvrez le fichier *Error.cshtml.cs* dans le dossier *Pages*. `ErrorViewModel` contient le code suivant :

```
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
```

Ajouter la directive `#nullable enable` avant la déclaration de classe et une directive `#nullable restore` après.

Vous obtiendrez un avertissement indiquant que `RequestId` n'est pas initialisé. En examinant la classe, vous devez décider que la propriété `RequestId` doit avoir la valeur null dans certains cas. L'existence de la propriété `ShowRequestId` indique que des valeurs manquantes sont possibles. Étant donné que `null` est valide, ajoutez `?` sur le type `string` pour indiquer que la propriété `RequestId` est un *type de référence nullable*. La classe finale se présente comme suit :

```
#nullable enable
    public class ErrorModel : PageModel
    {
        public string? RequestId { get; set; }

        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

        public void OnGet()
        {
            RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        }
    }
#nullable restore
```

Vérifiez les utilisations de la propriété, et vous voyez que dans la page associée, la présence de la valeur null est vérifiée dans la propriété avant de la restituer dans le balisage. C'est une utilisation sécurisée d'un type de référence nullable, donc vous en avez terminé avec cette classe.

La correction des valeurs null entraîne une modification

Fréquemment, le correctif pour un ensemble d'avertissements crée de nouveaux avertissements dans le code connexe. Nous allons voir les avertissements en action en corrigeant la classe `index.cshtml.cs`. Ouvrez le fichier `index.cshtml.cs` et examinez le code. Ce fichier contient le code sous-jacent à la page d'index :

```
public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a
syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}
```

Ajoutez la directive `#nullable enable` et vous verrez deux avertissements. Ni la propriété `ErrorText` ni la propriété `NewsItems` ne sont initialisées. Un examen de cette classe vous amènerait à croire que les deux propriétés devraient être des types de référence nuls : les deux ont des setters privés. Une seule exactement est attribuée dans la méthode `OnGet`. Avant d'apporter des modifications, examinez les consommateurs de ces deux propriétés. Dans la page elle-même, `ErrorText` est comparé à la valeur null avant de générer le balisage d'éventuelles erreurs. La collection `NewsItems` est vérifiée par rapport à `null` et afin de garantir que la collection comporte des éléments. Un correctif rapide serait d'attribuer le type de référence nullable aux deux propriétés. Un meilleur correctif consisterait à attribuer un type de référence non nullable à la collection et d'ajouter des éléments

à la collection existante lors de la récupération des articles de presse. Le premier correctif consiste à ajouter `?` au type `string` pour `ErrorText` :

```
public string? ErrorText { get; private set; }
```

Cette modification ne se propage pas à d'autre code, étant donné que tout accès à la propriété `ErrorText` a été déjà protégé par les contrôles de la valeur null. Ensuite, initialisez la liste `NewsItems` et supprimez la méthode setter de propriété, pour en faire une propriété en lecture seule :

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

Cela a résolu l'avertissement, mais a introduit une erreur. La liste `NewsItems` est désormais **correcte par construction**, mais le code qui définit la liste dans `OnGet` doit changer pour correspondre à la nouvelle API. Au lieu d'une attribution, appelez `AddRange` pour ajouter les éléments d'articles de presse à la liste existante :

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

Utiliser `AddRange` au lieu d'une attribution signifie que la méthode `GetNews` peut retourner `IEnumerable` au lieu de `List`. Cela permet d'économiser une attribution. Modifiez la signature de la méthode et supprimez l'appel `ToList`, comme indiqué dans l'exemple de code suivant :

```
public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
           new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}
```

La modification de la signature rompt aussi l'un des tests. Ouvrez le fichier `NewsServiceTests.cs` dans le dossier `Services` du projet `SimpleFeedReader.Tests`. Accédez au test `Returns_News_Stories_Given_Valid_Uri` et modifiez le type de la variable `result` sur `IEnumerable<NewsItem>`. La modification du type signifie que la propriété `Count` n'est plus disponible, par conséquent, remplacez la propriété `Count` dans `Assert` avec un appel à `Any()` :

```
// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());
```

Vous devrez ajouter une instruction `using System.Linq` au début du fichier également.

Cet ensemble de modifications met en évidence une attention particulière lors de la mise à jour de code qui inclut des instanciations génériques. La liste et les éléments dans la liste sont de type non nullable. L'un ou les deux peuvent être de type nullable. Toutes les déclarations suivantes sont autorisées :

- `List<NewsStoryViewModel>` : liste non nullable de modèles d'affichage non nullable.
- `List<NewsStoryViewModel?>` : liste non nullable de modèles d'affichage nullable.
- `List<NewsStoryViewModel>?` : liste nullable de modèles d'affichage non nullable.
- `List<NewsStoryViewModel?>?` : liste nullable de modèles d'affichage nullable.

Interfaces avec du code externe

Vous avez apporté des modifications à la classe `NewsService`, donc activez l'annotation `#nullable enable` pour cette classe. Cela ne génère aucun nouvel avertissements. Toutefois, l'examen attentif de la classe permet d'illustrer certaines des limitations de l'analyse du flux du compilateur. Examinez le constructeur :

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

Le paramètre `IMapper` est typé comme référence non nullable. Il est appelé par le code d'infrastructure ASP.NET Core, donc le compilateur ne sait pas vraiment que `IMapper` ne sera jamais null. Le conteneur d'injection de dépendance (DI) ASP.NET Core par défaut lève une exception s'il ne peut pas résoudre un service nécessaire, afin que le code soit correct. Le compilateur ne peut pas valider tous les appels à vos API publiques, même si votre code est compilé avec des contextes d'annotation nullable activés. En outre, vos bibliothèques peuvent être utilisées par des projets qui n'utilisent pas encore des types de référence nullable. Validez les entrées dans les API publiques même si vous les avez déclarées en tant que types non nullable.

Obtenir le code

Vous avez résolu les avertissements que vous avez identifiés dans la compilation du test initial, donc vous pouvez désormais activer le contexte d'annotation nullable pour les deux projets. Régénérez les projets ; le compilateur ne signale aucun avertissement. Vous pouvez obtenir le code pour le projet terminé dans le référentiel GitHub dotnet/samples.

Les nouvelles fonctionnalités qui prennent en charge les types de référence nullable vous aident à trouver et à corriger les erreurs potentielles dans la façon dont vous gérez les valeurs `null` dans votre code. L'activation du contexte d'annotation nullable vous permet d'exprimer votre intention de conception : certaines variables ne doivent jamais être null, d'autres variables peuvent contenir des valeurs null. Ces fonctionnalités facilitent la déclaration de votre intention de conception. De même, le contexte d'avertissement nullable indique au

compilateur d'émettre des avertissements lorsque vous avez enfreint cette intention. Ces avertissements vous guident pour effectuer des mises à jour qui rendent votre code plus robuste et moins susceptible de lever une `NullReferenceException` pendant l'exécution. Vous pouvez contrôler l'étendue de ces contextes afin de vous concentrer sur des zones locales de code à migrer alors que la base de code restante reste intacte. Dans la pratique, vous pouvez intégrer cette tâche de migration dans une maintenance régulière de vos classes. Ce tutoriel a présenté le processus de migration d'une application pour utiliser des types de référence nullable. Vous pouvez explorer un exemple réel plus complet de ce processus en examinant la demande de tirage effectuée par Jon Skeet pour incorporer les types de référence nullable dans NodaTime. Ou tout simplement En outre, vous pouvez apprendre des techniques pour l'utilisation de types de référence nuls avec Entity Framework Core dans Entity Framework Core - Travailler avec des types de référence nuls.

Didacticiel : générer et utiliser des flux asynchrones à l'aide de C# 8,0 et .NET Core 3,0

20/05/2020 • 17 minutes to read • Edit Online

C# 8,0 introduit des **flux asynchrones**, qui modélisent une source de données de diffusion en continu. Les flux de données récupèrent et génèrent souvent des éléments de façon asynchrone. Les flux asynchrones s'appuient sur les nouvelles interfaces introduites dans .NET Standard 2,1. Ces interfaces sont prises en charge dans .NET Core 3,0 et versions ultérieures. Ils fournissent un modèle de programmation naturel pour les sources de données de streaming asynchrones.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- créer une source de données qui génère une séquence d'éléments de données de façon asynchrone ;
- consommer cette source de données de façon asynchrone ;
- Prendre en charge l'annulation et les contextes capturés pour les flux asynchrones.
- reconnaître quand l'interface et la source de données nouvelles sont préférables aux séquences de données synchrones précédentes.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET Core, y compris le compilateur C# 8,0. Le compilateur C# 8 est disponible à partir de Visual Studio 2019 version 16,3 ou du Kit de développement logiciel (SDK) .net Core 3,0.

Vous devrez créer un jeton d'accès GitHub afin de pouvoir accéder au point de terminaison GitHub GraphQL. Sélectionnez les autorisations suivantes pour votre jeton d'accès GitHub :

- repo:status
- public_repo

Enregistrez le jeton d'accès à un endroit sûr afin de pouvoir l'utiliser pour accéder au point de terminaison de l'API GitHub.

> **WARNING**
> Sécurisez votre jeton d'accès personnel. Tous les logiciels disposant de votre jeton d'accès personnel peuvent effectuer des appels d'API GitHub à l'aide de vos droits d'accès.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET Core.

Exécutez l'application de démarrage

Vous pouvez obtenir le code pour l'application de démarrage utilisée dans ce didacticiel à partir du dépôt dotnet/docs dans le dossier CSharp/tutoriels/AsyncStreams .

L'application de démarrage est une application console qui utilise l'interface GitHub GraphQL pour récupérer des problèmes récents écrits dans le référentiel dotnet/docs. Commencez par examiner le code suivant pour la méthode `Main` de l'application de démarrage :

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-
token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

Vous pouvez soit définir une variable d'environnement `GitHubKey` sur votre jeton d'accès personnel, soit remplacer le dernier argument dans l'appel par `GenEnvVariable` avec votre jeton d'accès personnel. Ne placez pas votre code d'accès dans le code source si vous souhaitez partager la source avec d'autres utilisateurs. Ne téléchargez jamais de codes d'accès dans un référentiel source partagé.

Après la création du client de GitHub, le code dans `Main` crée un objet de rapport de progression et un jeton d'annulation. Une fois que ces objets sont créés, `Main` appelle `runPagedQueryAsync` pour récupérer les 250 problèmes créés les plus récents. Une fois cette tâche terminée, les résultats sont affichés.

Lorsque vous exécutez l'application de démarrage, vous pouvez faire quelques observations importantes concernant son fonctionnement. Vous voyez la progression signalée pour chaque page retournée à partir de GitHub. Vous pouvez observer un temps de pause avant le retour de chaque nouvelle page de problèmes par GitHub. Enfin, les problèmes ne sont affichés que lorsque les 10 pages ont été récupérées à partir de GitHub.

Examinez l'implémentation

L'implémentation révèle pourquoi vous avez observé le comportement décrit dans la section précédente. Examinez the code for `runPagedQueryAsync` :

```
private static async Task<JArray> runPagedQueryAsync(GitHubClient client, string queryText, string repoName,
CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Concentrons-nous sur l'algorithme de pagination et sur la structure asynchrone du code précédent. (Vous pouvez consulter la documentation de GitHub GraphQL pour plus d'informations sur l'API GraphQL github.) La `runPagedQueryAsync` méthode énumère les problèmes du plus récent au plus ancien. Elle a besoin de 25 problèmes par page et examine la structure `pageInfo` de la réponse pour continuer avec la page précédente. Cela suit la prise en charge standard de la pagination de GraphQL pour les réponses multipages. La réponse inclut un objet `pageInfo` qui contient une valeur `hasPreviousPages` et une valeur `startCursor` utilisées pour demander la page précédente. Les problèmes se trouvent dans le tableau `nodes`. La méthode `runPagedQueryAsync` ajoute ces nœuds à un tableau qui contient tous les résultats de toutes les pages.

Après la récupération et la restauration d'une page de résultats, `runPagedQueryAsync` signale la progression et vérifie l'annulation. Si l'annulation a été demandée, `runPagedQueryAsync` lève une OperationCanceledException.

Plusieurs éléments de ce code peuvent être améliorés. Plus important encore, `runPagedQueryAsync` doit allouer du stockage pour tous les problèmes retournés. Cet exemple s'arrête à 250 problèmes, car la récupération de tous les problèmes ouverts nécessiterait beaucoup plus de mémoire pour stocker tous les problèmes récupérées. Les protocoles de prise en charge des rapports de progression et de l'annulation rendent l'algorithme plus difficile à comprendre lors de sa première lecture. D'autres types et API sont impliqués. Vous devez suivre les communications via le CancellationTokenSource et son associé CancellationToken pour comprendre où l'annulation est demandée et où elle est accordée.

Les flux asynchrones sont mieux adaptés

Les flux asynchrones et la prise en charge associée du langage résolvent tous ces problèmes. Le code qui génère la

séquence peut désormais utiliser `yield return` pour retourner des éléments dans une méthode qui a été déclarée avec le modificateur `async`. Vous pouvez consommer un flux asynchrone à l'aide une boucle `await foreach` tout comme vous consommez n'importe quelle séquence à l'aide d'une boucle `foreach`.

Ces nouvelles fonctionnalités de langage dépendent de trois nouvelles interfaces ajoutées à .NET Standard 2.1 et implémentées dans .NET Core 3.0 :

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Ces trois interfaces sont très certainement familières à la plupart des développeurs C#. Elles se comportent de manière similaire à leurs équivalents synchrones :

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Il est possible que le type System.Threading.Tasks.ValueTask ne soit pas familier. Le struct `ValueTask` fournit une API similaire à la classe System.Threading.Tasks.Task. `ValueTask` est utilisé dans ces interfaces pour des raisons de performances.

Convertir en flux asynchrones

Ensuite, convertissez la méthode `runPagedQueryAsync` pour générer un flux asynchrone. Tout d'abord, modifiez la signature de `runPagedQueryAsync` pour retourner un `IAsyncEnumerable<JToken>` et supprimer les objets de jeton d'annulation et de progression de la liste de paramètres comme indiqué dans le code suivant :

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

Le code de démarrage traite chaque page lorsqu'elle est récupérée, comme indiqué dans le code suivant :

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Remplacez ces trois lignes par le code suivant :

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

Vous pouvez également supprimer la déclaration de `finalResults` plus tôt dans cette méthode et l'instruction `return` qui suit la boucle que vous avez modifiée.

Vous avez terminé les modifications permettant de générer un flux asynchrone. La méthode terminée doit ressembler au code suivant :

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Ensuite, vous modifiez le code qui utilise la collection pour consommer le flux de données asynchrone. Recherchez dans `Main` le code suivant, qui traite l'ensemble des problèmes :

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Remplacez-le par la boucle `await foreach` suivante :

```
int num = 0;
await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

La nouvelle interface IAsyncEnumerator<T> dérive de IAsyncDisposable . Cela signifie que la boucle précédente disposera de façon asynchrone le flux lorsque la boucle se terminera. Vous pouvez imaginer que la boucle ressemble au code suivant :

```
int num = 0;
var enumerator = runPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Par défaut, les éléments de flux sont traités dans le contexte capturé. Si vous souhaitez désactiver la capture du contexte, utilisez la TaskAsyncEnumerableExtensions.ConfigureAwait méthode d'extension. Pour plus d'informations sur les contextes de synchronisation et la capture du contexte actuel, consultez l'article sur l' utilisation du modèle asynchrone basésur des tâches.

Les flux asynchrones prennent en charge l'annulation en utilisant le même protocole que les autres `async` méthodes. Vous pouvez modifier la signature de la méthode d'itérateur Async comme suit pour prendre en charge l'annulation :

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

L' EnumeratorCancellationAttribute attribut force le compilateur à générer du code pour le IAsyncEnumerator<T> qui rend le jeton passé `GetAsyncEnumerator` visible au corps de l'itérateur Async comme cet argument. Dans `runQueryAsync` , vous pouvez examiner l'état du jeton et annuler un travail supplémentaire si nécessaire.

Vous utilisez une autre méthode d'extension, WithCancellation , pour transmettre le jeton d'annulation au flux asynchrone. Vous pouvez modifier la boucle en énumérant les problèmes comme suit :

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Vous pouvez obtenir le code du didacticiel terminé à partir du dépôt dotnet/docs dans le dossier CSharp/tutoriels/AsyncStreams .

Exécutez l'application terminée

Exécutez de nouveau l'application. Comparez son comportement avec le comportement de l'application de démarrage. La première page de résultats est énumérée dès qu'elle est disponible. Une pause peut être observée lorsque chaque nouvelle page est demandée et récupérée, puis les résultats de la page suivante sont rapidement

énumérés. Le bloc `try` / `catch` n'est pas nécessaire pour gérer l'annulation : l'appelant peut arrêter l'énumération de la collection. Le rapport de progression est clair, car le flux asynchrone génère des résultats à mesure que chaque page est téléchargée. L'état de chaque problème renvoyé est inclus en toute transparence dans la `await foreach` boucle. Vous n'avez pas besoin d'un objet de rappel pour suivre la progression.

Vous pouvez voir des améliorations lors de l'utilisation de mémoire en examinant le code. Vous n'avez plus besoin d'allouer une collection pour stocker tous les résultats avant qu'ils ne soient énumérés. L'appelant peut déterminer comment utiliser les résultats et si une collection de stockage est nécessaire.

Exécutez les applications de démarrage et les applications terminées. Ceci vous permettra d'observer les différences entre les implémentations pour vous. À la fin de ce tutoriel, vous pouvez supprimer le jeton d'accès GitHub que vous avez créé au début. Si un attaquant arrive à accéder à ce jeton, il pourrait accéder aux API GitHub à l'aide de vos informations d'identification.

Didacticiel : utiliser des critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données.

16/12/2020 • 27 minutes to read • Edit Online

C# 7 a introduit des fonctionnalités de critères spéciaux de base. Ces fonctionnalités sont étendues en C# 8 et C# 9 avec de nouveaux modèles et expressions. Il est possible d'écrire des fonctionnalités qui se comportent comme si des types provenant potentiellement d'autres bibliothèques avaient été étendus. Une autre utilisation des modèles consiste à créer des fonctionnalités requises par une application qui ne sont pas essentielles pour le type étendu.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- Reconnaître les situations dans lesquelles les critères spéciaux sont nécessaires.
- Utiliser des expressions de critères spéciaux pour implémenter des comportements en fonction des types et des valeurs de propriété.
- Combiner des critères spéciaux avec d'autres techniques pour créer des algorithmes complets.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 5, qui comprend le compilateur C# 9. Le compilateur C# 9 est disponible à partir de Visual Studio 2019 version 16,9 Preview 1 ou .net 5,0 SDK.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET Core.

Scénarios des critères spéciaux

Le développement moderne passe souvent par l'intégration de données provenant de plusieurs sources et par la présentation d'informations et d'insights tirés de ces données dans une seule et même application. Ni vous ni votre équipe n'aurez accès aux types qui représentent les données entrantes ou ne pourrez les contrôler.

Une conception classique orientée objet exigerait de créer dans l'application des types représentant chacun des types de données de ces multiples sources. L'application s'appuierait alors sur ces nouveaux types pour élaborer des hiérarchies d'héritage, créer des méthodes virtuelles et implémenter des abstractions. Ces techniques fonctionnent et représentent parfois les meilleurs outils disponibles. Il est possible dans d'autres cas d'écrire du code moins long et plus clair, à l'aide de techniques qui séparent les données des opérations qui les manipulent.

Ce tutoriel vise à créer et à explorer une application qui prend en entrée des données provenant de plusieurs sources externes pour un seul scénario. Nous verrons en quoi les **critères spéciaux** constituent un moyen efficace de consommer et de traiter ces données d'une autre manière que ce que prévoit le système d'origine.

Prenons une grande zone métropolitaine qui utilise des péages et des tarifs heures creuses/heures de pointe pour réguler le trafic. Nous allons écrire une application qui calcule le péage en fonction du type de véhicule. Lors d'améliorations ultérieures, nous intégrerons aux tarifs le nombre de passagers du véhicule ainsi que l'heure et le jour de la semaine.

À partir de cette brève description, vous avez peut-être rapidement esquissé une hiérarchie d'objets pour modéliser ce système. Cependant, les données proviennent de plusieurs sources, et notamment d'autres systèmes de gestion de l'immatriculation des véhicules. Ils fournissent des classes différentes pour modéliser ces données, sans proposer de modèle objet unique. Dans ce tutoriel, nous allons utiliser ces classes simplifiées pour modéliser les données sur les véhicules issues de ces systèmes externes, comme dans le code suivant :

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Vous pouvez télécharger l'exemple de démarrage à partir du référentiel GitHub dotnet/samples. Comme on peut le voir, les classes de véhicules proviennent de différents systèmes et utilisent différents espaces de noms. Aucune classe de base commune, autre que `System.Object`, n'est exploitable.

Conception avec les critères spéciaux

Le scénario utilisé dans ce didacticiel met en évidence les types de problèmes que la correspondance de modèle est bien adaptée pour résoudre :

- Les objets dont vous avez besoin ne figurent pas dans une hiérarchie d'objets correspondant à vos objectifs. Vous utilisez peut-être des classes faisant partie de systèmes non liés les uns aux autres.
- Les fonctionnalités ajoutées ne font pas partie de l'abstraction de base de ces classes. Le péage payé *change* selon les types de véhicules, mais il ne s'agit pas d'une fonction essentielle du véhicule.

Lorsque la *forme* des données et les *opérations* effectuées sur celles-ci ne sont pas décrites ensemble, les fonctionnalités de critères spéciaux de C# facilitent la tâche.

Implémenter les calculs de péage de base

Le calcul de péage le plus élémentaire s'appuie uniquement sur le type de véhicule :

- `Car` : 2,00 $.
- `Taxi` : 3,50 $.
- `Bus` : 5,00 $.
- `DeliveryTruck` : 10,00 $.

Créez une classe `TollCalculator` et implémentez les critères spéciaux sur le type de véhicule pour obtenir le montant du péage. Le code suivant montre l'implémentation initiale de `TollCalculator`.

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    public class TollCalculator
    {
        public decimal CalculateToll(object vehicle) =>
            vehicle switch
        {
            Car c           => 2.00m,
            Taxi t          => 3.50m,
            Bus b           => 5.00m,
            DeliveryTruck t => 10.00m,
            { }             => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
            null            => throw new ArgumentNullException(nameof(vehicle))
        };
    }
}
```

Le code précédent utilise une **expression switch** (différente d'une instruction `switch`) qui teste le **modèle de type**. Une **expression switch** commence par la variable, `vehicle` dans le code précédent, suivie du mot clé `switch`. Viennent ensuite toutes les **branches switch** entre accolades. L'expression `switch` ajoute d'autres perfectionnements à la syntaxe qui entoure l'instruction `switch`. Le mot clé `case` est omis, et le résultat de chaque branche est une expression. Les deux dernières montrent une nouvelle fonctionnalité du langage. Le cas `{ }` représente n'importe quel objet non Null ne correspondant à aucune des branches antérieures. Il intercepte tous les types incorrects passés à cette méthode. Le cas `{ }` doit suivre les cas de chaque type de véhicule. Si l'ordre était inversé, le cas `{ }` serait prioritaire. Enfin, le modèle `null` détecte si un `null` est passé à cette méthode. Le `null` modèle peut être en dernier, car les autres modèles de type correspondent uniquement à un objet non null du type correct.

Pour tester ce code, utilisez le code suivant dans `Program.cs` :

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            var tollCalc = new TollCalculator();

            var car = new Car();
            var taxi = new Taxi();
            var bus = new Bus();
            var truck = new DeliveryTruck();

            Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
            Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
            Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
            Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

            try
            {
                tollCalc.CalculateToll("this will fail");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Caught an argument exception when using the wrong type");
            }
            try
            {
                tollCalc.CalculateToll(null!);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("Caught an argument exception when using null");
            }
        }
    }
}
```

Ce code est inclus dans le projet de démarrage, mais il est commenté. Supprimez les commentaires et vous pouvez tester ce que vous avez écrit.

On commence à voir comment les modèles peuvent nous aider à créer des algorithmes dans lesquels le code et les données sont séparés. L' expression `switch` teste le type et produit différentes valeurs en fonction des résultats. Ce n'est que le début.

Ajouter la tarification en fonction du taux d'occupation

L'autorité de péage souhaite inciter les véhicules à circuler à capacité maximale. Elle a décidé de fixer un tarif plus élevé pour les véhicules transportant peu de passagers et un prix inférieur pour les véhicules pleins, de façon à encourager ces derniers :

- Voitures et taxis sans passager : +0,50 $.
- Voitures et taxis avec deux passagers : remise de 0,50 $.
- Voitures et taxis avec trois passagers ou plus : -1,00 $.
- Bus remplis à moins de 50 % : +2,00 $.
- Bus remplis à plus de 90 % : -1,00 $.

Il est possible d'implémenter ces règles avec le **modèle de propriété** dans la même expression switch. Le modèle de propriété examine les propriétés de l'objet une fois que le type a été déterminé. L'unique cas `Car` est étendu à quatre cas différents :

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car c                      => 2.00m - 1.0m,

    // ...
};
```

Les trois premiers cas testent le type `Car`, puis vérifient la valeur de la propriété `Passengers`. Si les deux correspondent, cette expression est évaluée et retournée.

Les cas des taxis sont étendus de la même manière :

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 3.50m - 1.00m,

    // ...
};
```

Dans l'exemple précédent, la clause `when` a été omise sur le cas final.

Maintenant, implémentons les règles de taux d'occupation en développant les cas des bus, comme dans l'exemple suivant :

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    // ...
};
```

L'autorité de péage ne se soucie pas du nombre de passagers des camions de livraison, et ajuste le montant en fonction du poids du camion, en procédant ainsi :

- Camions de plus de 5 000 lb : +5,00 $.
- Camions de moins de 3000 lb : -2,00 $.

Cette règle est implémentée dans le code suivant :

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};
```

Le code précédent montre la clause `when` d'une branche switch. Cette clause `when` sert à tester les conditions autres que l'égalité sur une propriété. Une fois que vous avez terminé, vous disposez d'une méthode qui ressemble au code suivant :

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car c                      => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

La plupart de ces branches switch sont des exemples de **modèles récursifs**. Par exemple, `Car { Passengers: 1}` indique un modèle de constante à l'intérieur d'un modèle de propriété.

Il est possible de rendre ce code moins répétitif avec des switch imbriqués. Dans les exemples précédents, `Car` et `Taxi` ont chacun quatre branches différentes. On peut créer dans les deux cas un modèle de type qui vient alimenter un modèle de propriété. Cette technique est illustrée dans le code suivant :

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

Dans l'exemple précédent, l'expression récursive évite de répéter les branches `Car` et `Taxi` contenant des branches enfant qui testent la valeur de propriété. Cette technique n'est pas utilisée pour les branches `Bus` et `DeliveryTruck`, car elles correspondent à des plages de test pour la propriété, et non à des valeurs discrètes.

Ajouter la tarification heures creuses/heures de pointe

En guise de fonctionnalité finale, l'autorité de péage souhaite ajouter une tarification soumise à une contrainte horaire. Pendant les heures d'affluence du matin et du soir, les péages sont doublés. Cette règle ne s'applique au trafic que dans un sens : en allant vers la ville le matin et en en sortant le soir. Le reste du temps, pendant la journée de travail, les péages augmentent de 50 %. Tard le soir et tôt le matin, ils sont réduits de 25 %. Le taux normal s'applique le weekend, quelle que soit l'heure. Vous pouvez utiliser une série si `if` les `else` instructions et permettent d'exprimer ce qui suit à l'aide du code suivant :

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        } else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

Le code précédent fonctionne correctement, mais n'est pas lisible. Vous devez chaîner tous les cas d'entrée et les instructions imbriquées `if` pour la raison du code. Au lieu de cela, vous allez utiliser des critères spéciaux pour cette fonctionnalité, mais vous allez l'intégrer à d'autres techniques. Vous pourriez élaborer une seule expression de critères spéciaux qui tiendrait compte de toutes les combinaisons direction/jour de la semaine/heure. Elle serait cependant complexe, difficile à lire, à comprendre et à vérifier. Combinons plutôt ces méthodes afin de créer un tuple de valeurs décrivant de manière concise tous ces états. Ensuite, utilisons les critères spéciaux pour calculer un multiplicateur de péage. Le tuple comporte trois conditions discrètes :

- le jour : semaine / weekend ;
- la tranche horaire ;
- la direction : vers l'intérieur ou l'extérieur de la ville.

Le tableau suivant montre les combinaisons de valeurs d'entrée et le multiplicateur tarifaire :

JOUR	TEMPS	SENS	PREMIUM
Jour de la semaine	Heure de pointe du matin	Vers l'intérieur de la ville	x 2,00
Jour de la semaine	Heure de pointe du matin	Vers l'extérieur de la ville	x 1,00
Jour de la semaine	Journée	Vers l'intérieur de la ville	x 1,50
Jour de la semaine	Journée	Vers l'extérieur de la ville	x 1,50
Jour de la semaine	Heure de pointe du soir	Vers l'intérieur de la ville	x 1,00
Jour de la semaine	Heure de pointe du soir	Vers l'extérieur de la ville	x 2,00
Jour de la semaine	Nuit	Vers l'intérieur de la ville	x 0,75
Jour de la semaine	Nuit	Vers l'extérieur de la ville	x 0,75
Week-end	Heure de pointe du matin	Vers l'intérieur de la ville	x 1,00
Week-end	Heure de pointe du matin	Vers l'extérieur de la ville	x 1,00
Week-end	Journée	Vers l'intérieur de la ville	x 1,00
Week-end	Journée	Vers l'extérieur de la ville	x 1,00
Week-end	Heure de pointe du soir	Vers l'intérieur de la ville	x 1,00
Week-end	Heure de pointe du soir	Vers l'extérieur de la ville	x 1,00
Week-end	Nuit	Vers l'intérieur de la ville	x 1,00
Week-end	Nuit	Vers l'extérieur de la ville	x 1,00

Les trois variables produisent 16 combinaisons différentes. On peut simplifier l'expression switch finale en associant certaines conditions.

Le système qui collecte les péages utilise une structure DateTime pour indiquer l'heure de collecte. Élaborons les méthodes de membre qui créent des variables à partir du tableau précédent. La fonction suivante utilise une expression switch de critères spéciaux pour exprimer si DateTime représente un jour de weekend ou de semaine :

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

Cette méthode est correcte, mais elle est répétitive. On peut la simplifier comme dans le code suivant :

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Ensuite, ajoutons une fonction similaire pour catégoriser l'heure dans les blocs :

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

Vous ajoutez un privé `enum` pour convertir chaque plage horaire en valeur discrète. Ensuite, la `GetTimeBand` méthode utilise des *modèles relationnels*, et des *conjonctives ou des modèles*, tous deux ajoutés en C# 9,0. Le modèle relationnel vous permet de tester une valeur numérique à l'aide `<` de,, `>` `<=` ou `>=` . Le `or` modèle teste si une expression correspond à un ou plusieurs modèles. Vous pouvez également utiliser un `and` modèle pour vous assurer qu'une expression correspond à deux modèles distincts et un `not` modèle pour vérifier qu'une expression ne correspond pas à un modèle.

Maintenant que nous avons créé ces méthodes, utilisons une autre expression `switch` avec le **modèle de tuple** pour calculer le multiplicateur tarifaire. On pourrait concevoir une expression `switch` comportant les 16 branches :

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

Le code ci-dessus fonctionne, mais on peut le simplifier. Les huit combinaisons du weekend correspondent au

même péage. Elles sont remplaçables par la ligne suivante :

```
(false, _, _) => 1.0m,
```

Le trafic entrant et le trafic sortant ont le même multiplicateur pendant la journée en semaine et pendant la nuit. Ces quatre branches switch peuvent être remplacées par les deux lignes suivantes :

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

Après ces deux modifications, le code devrait se présenter ainsi :

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Enfin, on peut supprimer les deux branches des heures de pointe au tarif normal, ce qui permet de remplacer `false` par un discard (`_`) dans la branche switch finale. On obtient la méthode suivante une fois terminée :

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

Cet exemple met en évidence un des avantages des critères spéciaux : les branches du modèle sont évaluées dans l'ordre. Si vous les réorganisez de telle sorte qu'une branche antérieure gère un des cas suivants, le compilateur vous avertit que le code est inaccessible. Grâce à ces règles de langage, nous avons pu effectuer facilement les simplifications précédentes sans craindre de modifier le code.

Les critères spéciaux rendent certains types de code plus lisibles et offrent une alternative aux techniques orientées objet quand vous ne pouvez pas ajouter de code à vos classes. Le cloud est à l'origine d'une séparation entre les données et les fonctionnalités. La *forme* des données et les *opérations* effectuées sur ces dernières ne sont pas nécessairement décrites ensemble. Dans ce tutoriel, nous avons exploité les données existantes d'une manière totalement différente de leur fonction d'origine. Les critères spéciaux offrent la possibilité d'écrire des fonctionnalités qui remplacent ces types, même sans pouvoir les étendre.

Étapes suivantes

Vous pouvez télécharger le code terminé dans référentiel GitHub dotnet/samples. Explorez les modèles par vous-même et ajoutez cette technique à vos activités de codage régulières. Ces techniques représentent une autre façon d'aborder les problèmes et de créer de nouvelles fonctionnalités.

Application console

02/11/2020 • 23 minutes to read • Edit Online

Ce didacticiel vous présente un certain nombre de fonctionnalités de .NET Core et du langage C#. Vous apprendrez ce qui suit :

- Les principes de base du CLI .NET Core
- La structure d'une application de console C#
- E/S console
- Les principes fondamentaux des API d'E/S de fichier dans .NET
- Les principes fondamentaux de la programmation asynchrone basée sur des tâches dans .NET

Vous allez générer une application qui lit un fichier texte et renvoie le contenu de ce fichier texte sur la console. La sortie sur la console se fait à un rythme permettant de la lire à haute voix. Vous pouvez accélérer ou ralentir le rythme en appuyant sur les clés « < » (inférieure à) ou « > » (supérieur à).

Il existe un grand nombre de fonctionnalités dans ce didacticiel. Créons-les un par un.

Prérequis

- Configurez votre ordinateur pour exécuter .NET Core. Vous trouverez les instructions d'installation dans la page téléchargements .net Core . Vous pouvez exécuter cette application sur Windows, Linux, macOS ou dans un conteneur d'ancrage.
- Installez votre éditeur de code favori.

Créer l'application

La première étape consiste à créer une nouvelle application. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel. Saisissez la commande `dotnet new console` à l'invite. Elle crée les fichiers de démarrage d'une application « Hello World » de base.

Avant de commencer à apporter des modifications, suivez les étapes pour exécuter l'application simple Hello World. Après avoir créé l'application, saisissez `dotnet restore` à l'invite de commandes. Cette commande exécute le processus de restauration de package NuGet. NuGet est un gestionnaire de packages .NET. Cette commande télécharge les dépendances manquantes pour votre projet. Comme il s'agit d'un nouveau projet, aucune des dépendances n'est en place, donc la première exécution téléchargera le framework .NET Core. Après cette étape initiale, vous devrez exécuter `dotnet restore` lorsque vous ajoutez de nouveaux packages dépendants ou mettez à jour les versions de vos dépendances.

Vous n'avez pas à exécuter `dotnet restore` , car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration, comme,,,, `dotnet new` `dotnet build` `dotnet run` `dotnet test` `dotnet publish` et `dotnet pack` . Pour désactiver la restauration implicite, utilisez l' `--no-restore` option.

La `dotnet restore` commande est toujours utile dans certains scénarios où la restauration explicite est appropriée, comme les Builds d'intégration continue dans Azure DevOps services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

Pour plus d'informations sur la gestion des flux NuGet, consultez la `dotnet restore` documentation.

Après la restauration des packages, vous exécutez `dotnet build`. Cela exécute le moteur de génération et crée l'exécutable de votre application. Enfin, vous exécutez `dotnet run` pour lancer votre application.

Le code d'application Hello World simple se trouve intégralement dans le fichier Program.cs. Ouvrez ce fichier avec votre éditeur de texte préféré. Nous allons apporter nos premières modifications. En haut du fichier, remarquez l'instruction using :

```
using System;
```

Cette instruction indique au compilateur que tous les types de l'espace de noms `System` sont dans la portée. Comme d'autres langages orientés objet que vous avez peut-être utilisés, C# utilise des espaces de noms pour organiser les types. Ce programme Hello World ne fait pas exception. Vous pouvez voir que le programme est placé dans l'espace de noms nommé d'après le nom du répertoire actuel. Pour ce tutoriel, nous changeons le nom de l'espace de noms en `TeleprompterConsole` :

```
namespace TeleprompterConsole
```

Lecture et affichage du fichier

La première fonctionnalité à ajouter est la capacité à lire un fichier texte et à afficher tout le texte dans la console. Tout d'abord, nous allons ajouter un fichier texte. Copiez le fichier sampleQuotes.txt à partir du dépôt GitHub pour cet exemple dans votre répertoire de projet. Il servira de script pour votre application. Si vous voulez des informations sur la façon de télécharger l'exemple d'application de cette rubrique, consultez les instructions de la rubrique Exemples et didacticiels.

Ensuite, ajoutez la méthode suivante dans votre classe `Program` (juste en dessous de la méthode `Main`) :

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Cette méthode utilise les types de deux nouveaux espaces de noms. Pour compiler, vous devez ajouter les deux lignes suivantes au début du fichier :

```
using System.Collections.Generic;
using System.IO;
```

L'interface IEnumerable<T> est définie dans l'espace de noms System.Collections.Generic. La classe File est définie dans l'espace de noms System.IO.

Cette méthode est un type spécial de méthode C# appelé *méthode d'itérateur*. Les méthodes d'énumérateur retournent des séquences qui sont évaluées de manière tardive. Cela signifie que chaque élément de la séquence est généré lorsque cela est demandé par le code utilisant la séquence. Les méthodes d'énumérateur sont des méthodes qui contiennent une ou plusieurs `yield return` instructions. L'objet retourné par la méthode `ReadFrom` contient le code pour générer chaque élément dans la séquence. Dans cet exemple, cela implique la lecture de la ligne de texte suivante à partir du fichier source et le renvoi de cette chaîne. Chaque fois que le code appelant demande l'élément suivant de la séquence, le code lit la ligne suivante du texte à partir du fichier et la renvoie. Quand le fichier est entièrement lu, la séquence indique qu'il n'y a pas d'autres éléments.

Il existe deux autres éléments de syntaxe de C# qui peuvent être nouveaux pour vous. L' `using` instruction dans cette méthode gère le nettoyage des ressources. La variable initialisée dans l'instruction `using` (`reader` , dans cet exemple) doit implémenter l'interface IDisposable. Cette interface définit une méthode unique, `Dispose` , qui doit être appelée lorsque les ressources doivent être libérées. Le compilateur génère l'appel lorsque l'exécution atteint l'accolade fermante de l'instruction `using` . Le code généré par le compilateur garantit que la ressource est libérée même si une exception est levée à partir du code dans le bloc défini par l'instruction using.

La variable `reader` est définie à l'aide du mot-clé `var` . `var` définit une *variable locale implicitement typée*. Cela signifie que le type de la variable est déterminé par le type au moment de la compilation de l'objet assigné à la variable. Ici, c'est la valeur retournée par la méthode OpenText(String), qui est un objet StreamReader.

À présent, nous allons renseigner le code pour lire le fichier dans la `Main` méthode :

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Exécutez le programme (à l'aide de `dotnet run`). Chaque ligne est envoyée sur la console.

Ajout de délais et de mise en forme à la sortie

Ce que vous avez s'affiche beaucoup trop rapidement pour le lire à haute voix. Vous devez maintenant ajouter des délais à la sortie. Au démarrage, vous allez générer un code de base qui permet un traitement asynchrone. Toutefois, ces premières étapes suivront quelques anti-modèles. Les anti-modèles sont signalés dans les commentaires lorsque vous ajoutez le code, et le code sera actualisé ultérieurement.

Il existe deux étapes dans cette section. Tout d'abord, vous allez mettre à jour la méthode Iterator pour retourner des mots uniques au lieu de lignes entières. Cela se fait avec ces modifications. Remplacez l'instruction `yield return line;` par le code suivant :

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Ensuite, vous devez modifier la façon dont vous consommez les lignes du fichier, et ajoutez un délai après chaque mot. Remplacez l'instruction `Console.WriteLine(line)` dans la méthode `Main` avec le bloc suivant :

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

La classe Task se trouve dans l'espace de noms System.Threading.Tasks, vous devez donc ajouter cette instruction `using` au début du fichier :

```
using System.Threading.Tasks;
```

Exécutez l'exemple et vérifiez le résultat. À présent, chaque mot unique est affiché séparément, suivi d'un délai de 200 ms. Toutefois, la sortie affichée présente certains problèmes, car le fichier texte source possède plusieurs lignes qui ont plus de 80 caractères sans saut de ligne. Ce qui peut être difficile à lire lors du défilement. C'est facile à corriger. Il vous suffit d'effectuer le suivi de la longueur de chaque ligne et de générer une nouvelle ligne chaque fois que la longueur de la ligne atteint un certain seuil. Déclarez une variable locale après la déclaration de `words` dans la méthode `ReadFrom` qui contient la longueur de ligne :

```
var lineLength = 0;
```

Ensuite, ajoutez le code suivant après l'instruction `yield return word + " ";` (avant l'accolade fermante) :

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Exécutez l'exemple et vous serez en mesure de lire à haute voix à son rythme préconfiguré.

Tâches asynchrones

Dans cette étape finale, vous allez ajouter le code pour écrire la sortie de façon asynchrone dans une tâche, tout en exécutant également une autre tâche pour lire l'entrée de l'utilisateur s'il souhaite accélérer ou ralentir l'affichage du texte, ou arrêter complètement l'affichage du texte. Il y a quelques étapes dans l'informatique et, à la fin, vous disposerez de toutes les mises à jour dont vous avez besoin. La première étape consiste à créer une Task méthode de retour asynchrone qui représente le code que vous avez créé jusqu'à présent pour lire et afficher le fichier.

Ajoutez cette méthode à votre `Program` classe (elle est extraite du corps de votre `Main` méthode) :

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Vous remarquerez deux modifications. Tout d'abord, dans le corps de la méthode, au lieu d'appeler Wait() pour attendre de manière synchrone qu'une tâche se termine, cette version utilise le mot-clé `await`. Pour ce faire, vous devez ajouter le modificateur `async` pour la signature de méthode. Cette méthode renvoie une `Task`. Notez qu'il n'y a aucune instruction de retour renvoyant un objet `Task`. Au lieu de cela, cet objet `Task` est créé par le code que le compilateur génère lorsque vous utilisez l'opérateur `await`. Vous pouvez imaginer que cette méthode renvoie une valeur lorsqu'elle atteint un `await`. La `Task` renvoyée indique que la tâche n'a pas été effectuée. La méthode reprend lorsque la tâche attendue se termine. Lorsqu'elle a terminé son exécution, la `Task` renvoyée indique qu'elle est terminée. Le code appelant peut surveiller cette `Task` renvoyée pour déterminer si elle est terminée.

Vous pouvez appeler cette nouvelle méthode dans votre méthode `Main` :

```
ShowTeleprompter().Wait();
```

Ici, dans `Main`, le code attend de façon synchrone. Vous devez utiliser l'opérateur `await` au lieu d'attendre de manière synchrone lorsque cela est possible. Toutefois, dans la méthode d'une application console `Main` , vous ne pouvez pas utiliser l' `await` opérateur. Cela entraînerait la fermeture de l'application avant la fin de toutes les tâches.

> **NOTE**
> Si vous utilisez C# 7.1 ou une version ultérieure, vous pouvez créer des applications de console à l'aide de la méthode `async` `Main`.

Ensuite, vous devez écrire la deuxième méthode asynchrone pour lire à partir de la console et surveiller les clés « < » (inférieur à), « > » (supérieur à) et « X » ou « x ». Voici la méthode que vous ajoutez pour cette tâche :

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

Cela crée une expression lambda pour représenter un Action délégué qui lit une clé à partir de la console et modifie une variable locale représentant le délai quand l'utilisateur appuie sur les clés « < » (inférieur à) ou « > » (supérieur à). La méthode déléguée se termine lorsque l'utilisateur appuie sur les touches « X » ou « x », ce qui permet à l'utilisateur d'arrêter l'affichage du texte à tout moment. Cette méthode utilise ReadKey() pour bloquer et attendre que l'utilisateur appuie sur une touche.

Pour terminer cette fonctionnalité, vous devez créer une nouvelle méthode de retour `async Task` qui démarre ces deux tâches (`GetInput` et `ShowTeleprompter`) et gère également les données partagées entre ces deux tâches.

Il est temps de créer une classe capable de gérer les données partagées entre ces deux tâches. Cette classe contient deux propriétés publiques : le délai et un indicateur `Done` pour indiquer que le fichier a été entièrement lu :

```
namespace TelepromptерConsole
{
    internal class TelePrompterConfig
    {
        public int DelayInMilliseconds { get; private set; } = 200;

        public void UpdateDelay(int increment) // negative to speed up
        {
            var newDelay = Min(DelayInMilliseconds + increment, 1000);
            newDelay = Max(newDelay, 20);
            DelayInMilliseconds = newDelay;
        }

        public bool Done { get; private set; }

        public void SetDone()
        {
            Done = true;
        }
    }
}
```

Placez cette classe dans un nouveau fichier et ajoutez-la à l'espace de noms `TelepromptерConsole`, comme indiqué ci-dessus. Vous devez également ajouter une `using static` instruction pour pouvoir référencer les `Min` `Max` méthodes et sans les noms de classe ou d'espace de noms englobants. Une `using static` instruction importe les méthodes d'une classe. Cela est en opposition avec les instructions `using` utilisées jusqu'à présent, et qui avaient importé toutes les classes à partir d'un espace de noms.

```
using static System.Math;
```

Ensuite, vous devez mettre à jour les méthodes `ShowTeleprompter` et `GetInput` pour utiliser le nouvel objet `config`. Écrivez une dernière `Task` renvoyant une méthode `async` pour démarrer les deux tâches et quitter lorsque la première tâche est terminée :

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

Ici, la nouvelle méthode est l'appel WhenAny(Task[]). Elle crée une `Task` qui se termine dès que les tâches de sa liste d'arguments se terminent.

Ensuite, vous devez mettre à jour les méthodes `ShowTeleprompter` et `GetInput` pour utiliser l'objet `config` pour le délai :

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

Cette nouvelle version de `ShowTeleprompter` appelle une nouvelle méthode dans la classe `TelepromptерConfig`. À présent, vous devez mettre à jour `Main` pour appeler `RunTeleprompter` au lieu de `ShowTeleprompter` :

```
RunTeleprompter().Wait();
```

Conclusion

Ce didacticiel vous a montré certaines des fonctionnalités du langage C# et les bibliothèques .NET Core liées au travail dans les applications console. Vous pouvez utiliser ces connaissances pour explorer davantage le langage ainsi que les classes présentées ici. Vous avez vu les principes de base des e/s de fichier et de console, l'utilisation de blocage et de non-blocage de la programmation asynchrone basée sur des tâches, une visite guidée du langage C# et de la façon dont les programmes C# sont organisés, et les CLI .NET Core.

Pour plus d'informations sur les E/S de fichier, consultez la rubrique E/S de fichier et de flux. Pour plus d'informations sur le modèle de programmation asynchrone utilisé dans ce tutoriel, consultez la rubrique Programmation asynchrone basée sur les tâches et la rubrique Programmation asynchrone.

Client REST

02/11/2020 • 22 minutes to read • Edit Online

Ce didacticiel vous présente un certain nombre de fonctionnalités de .NET Core et du langage C#. Vous apprendrez à :

- Notions de base du CLI .NET Core.
- Une présentation des fonctionnalités du langage C#.
- Gestion des dépendances avec NuGet
- Communications HTTP
- Traitement des informations JSON
- Gestion de la configuration avec les attributs.

Vous allez générer une application qui émet des requêtes HTTP vers un service REST sur GitHub. Vous lirez des informations au format JSON et convertirez ce paquet JSON en objets C#. Enfin, vous verrez comment travailler avec les objets C#.

Ce didacticiel présente de nombreuses fonctionnalités. Nous allons les construire une par une.

Si vous préférez utiliser l'exemple final pour cette rubrique, vous pouvez le télécharger. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET core. Vous trouverez les instructions d'installation dans la page téléchargements .net Core . Vous pouvez exécuter cette application sur Windows, Linux, Mac OS ou dans un conteneur Docker. Vous devez installer l'éditeur de code de votre choix. Les descriptions ci-dessous utilisent Visual Studio code, qui est un éditeur de plateformes Open source et multiplateforme. Cependant, vous pouvez utiliser les outils avec lesquels vous êtes le plus à l'aise.

Création de l'application

La première étape consiste à créer une nouvelle application. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel. Entrez la commande suivante dans une fenêtre de console :

```
dotnet new console --name WebAPIClient
```

Elle crée les fichiers de démarrage d'une application « Hello World » de base. Le nom du projet est « WebAPIClient ». Comme il s'agit d'un nouveau projet, aucune des dépendances n'est en place. La première exécution télécharge l'infrastructure .NET Core, installe un certificat de développement et exécute le gestionnaire de package NuGet pour restaurer les dépendances manquantes.

Avant de commencer à apporter des modifications, tapez `dotnet run` (voir la remarque) à l'invite de commandes pour exécuter votre application. `dotnet run` effectue automatiquement `dotnet restore` s'il manque des dépendances dans votre environnement. Il effectue également `dotnet build` si votre application doit être regénérée. Après la configuration initiale, vous devez uniquement exécuter `dotnet restore` ou `dotnet build` quand cela est pertinent pour votre projet.

Ajout de nouvelles dépendances

L'un des objectifs de conception clés de .NET Core consiste à réduire la taille de l'installation .NET. Si une application a besoin de bibliothèques supplémentaires pour certaines de ses fonctionnalités, vous ajoutez ces dépendances dans votre fichier projet (*.csproj) en C#. Pour notre exemple, vous devez ajouter le `System.Runtime.Serialization.Json` package, afin que votre application puisse traiter les réponses JSON.

Vous aurez besoin du `System.Runtime.Serialization.Json` package pour cette application. Ajoutez-le à votre projet en exécutant la commande CLI .net suivante :

```
dotnet add package System.Text.Json
```

Effectuer des requêtes web

Vous êtes maintenant prêt à commencer la récupération des données à partir du web. Dans cette application, vous obtiendrez des informations à partir de l'API GitHub. Nous allons lire des informations sur les projets couverts par la .NET Foundation. Vous allez commencer par créer la demande à l'API GitHub pour récupérer des informations sur les projets. Vous allez utiliser le point de terminaison https://api.github.com/orgs/dotnet/repos. Vous souhaitez récupérer toutes les informations sur ces projets, vous allez donc utiliser une requête HTTP GET. Votre navigateur utilise également les requêtes HTTP GET, vous pouvez donc coller cette URL dans votre navigateur pour voir les informations que vous recevez et traitez.

Vous utilisez la classe HttpClient pour effectuer des requêtes web. Comme toutes les API .NET modernes, HttpClient prend en charge uniquement les méthodes async pour ses API les plus anciennes. Commencez par créer une méthode async. Vous allez effectuer l'implémentation lors de la création des fonctionnalités de l'application. Commencez en ouvrant le fichier `program.cs` dans votre répertoire de projet et l'ajout de la méthode suivante à la classe `Program` :

```
private static async Task ProcessRepositories()
{
}
```

Vous devez ajouter une `using` directive en haut de votre `Main` méthode afin que le compilateur C# reconnaisse le Task type :

```
using System.Threading.Tasks;
```

Si vous générez votre projet à ce stade, vous obtiendrez un avertissement généré pour cette méthode, car elle ne contient aucun opérateur `await` et s'exécute de façon synchrone. Ignorez-le pour le moment. vous ajouterez des opérateurs `await` lorsque vous remplirez la méthode.

Ensuite, mettez à jour la `Main` méthode pour appeler la `ProcessRepositories` méthode. La `ProcessRepositories` méthode retourne une tâche et vous ne devez pas quitter le programme avant que cette tâche se termine. Par conséquent, vous devez modifier la signature de `Main` . Ajoutez le `async` modificateur, puis remplacez le type de retour par `Task` . Ensuite, dans le corps de la méthode, ajoutez un appel à `ProcessRepositories` . Ajoutez le `await` mot clé à cet appel de méthode :

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

À présent, vous disposez d'un programme qui ne fait rien, mais le fait de façon asynchrone. Nous allons l'améliorer.

Tout d'abord, vous avez besoin d'un objet pouvant récupérer des données à partir du web ; pour ce faire, vous pouvez utiliser HttpClient. Cet objet gère la demande et les réponses. Instanciez une instance unique de ce type dans la `Program` classe à l'intérieur du fichier *Program.cs* .

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}
```

Revenons à la méthode `ProcessRepositories` et créons sa première version :

```
private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

Pour compiler, vous devez également ajouter deux nouvelles `using` directives en haut du fichier :

```
using System.Net.Http;
using System.Net.Http.Headers;
```

Cette première version effectue une requête web pour lire la liste de tous les dépôts de l'organisation dotnet foundation. (L'ID GitHub de la .NET Foundation est 'dotnet'). Les premières lignes configurent HttpClient pour cette requête. Tout d'abord, il est configuré pour accepter les réponses JSON de GitHub. Ce format est simplement du JSON. La ligne suivante ajoute un en-tête d'agent utilisateur à toutes les requêtes à partir de cet objet. Ces deux en-têtes sont vérifiés par le code du serveur GitHub et sont nécessaires pour récupérer des informations à partir de GitHub.

Une fois que vous avez configuré le HttpClient, effectuez une requête web et récupérez la réponse. Dans cette première version, vous utilisez la méthode pratique HttpClient.GetStringAsync(String). Cette méthode démarre une tâche qui effectue la requête web, et, lorsque la demande est renvoyée, lit le flux de réponse puis extrait le contenu à partir du flux. Le corps de la réponse est renvoyé en tant que String. La chaîne est disponible lorsque la tâche est terminée.

Les deux dernières lignes de cette méthode attendent cette tâche et impriment la réponse dans la console. Générez l'application et exécutez-la. L'avertissement de génération a disparu maintenant, car `ProcessRepositories` contient maintenant un opérateur `await`. Vous verrez long affichage de texte au format JSON.

Traitement du résultat JSON

À ce stade, vous avez écrit le code pour récupérer une réponse à partir d'un serveur web et afficher le texte contenu dans cette réponse. Ensuite, nous allons convertir cette réponse JSON en objets C#.

La System.Text.Json.JsonSerializer classe sérialise les objets au format JSON et désérialise JSON en objets. Commencez par définir une classe pour représenter l' `repo` objet JSON retourné à partir de l'API GitHub :

```
using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}
```

Placez le code ci-dessus dans un fichier nommé « repo.cs ». Cette version de la classe représente le chemin d'accès le plus simple pour traiter les données JSON. Le nom de classe et le nom du membre correspondent aux noms utilisés dans le paquet JSON, au lieu des conventions C# suivantes. Vous allez corriger cela en fournissant des attributs de configuration ultérieurement. Cette classe décrit une autre fonctionnalité importante de la sérialisation et de désérialisation JSON : tous les champs dans le paquet JSON font partie de cette classe. Le sérialiseur JSON ignore les informations qui ne sont pas incluses dans le type de classe utilisé. Cette fonctionnalité facilite la création de types qui fonctionnent avec uniquement un sous-ensemble des champs dans le paquet JSON.

Maintenant que vous avez créé le type, nous allons le désérialiser.

Ensuite, vous utiliserez le sérialiseur pour convertir le JSON en objets C#. Remplacez l'appel à GetStringAsync(String) dans votre `ProcessRepositories` méthode par les lignes suivantes :

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

Vous utilisez de nouveaux espaces de noms. vous devez donc les ajouter en haut du fichier :

```
using System.Collections.Generic;
using System.Text.Json;
```

Notez que vous utilisez maintenant GetStreamAsync(String) au lieu de GetStringAsync(String). Le sérialiseur utilise un flux plutôt qu'une chaîne comme source. Nous allons expliquer quelques-unes des fonctionnalités du langage C# qui sont utilisées dans la deuxième ligne de l'extrait de code précédent. Le premier argument de JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) est une `await` expression. (Les deux autres paramètres sont facultatifs et sont omis dans l'extrait de code.) Les expressions await peuvent apparaître presque n'importe où dans votre code, même si vous ne les avez vu que dans le cadre d'une instruction d'assignation. La `Deserialize` méthode est *générique*, ce qui signifie que vous devez fournir des arguments de type pour les types d'objets qui doivent être créés à partir du texte JSON. Dans cet exemple, vous désérialisez vers un `List<Repository>` , qui est un autre objet générique, le System.Collections.Generic.List<T> . La `List<>` classe stocke une collection d'objets. L'argument de type déclare le type des objets stockés dans le `List<>` . Le texte JSON représente une collection d'objets référentiel, donc l'argument de type est `Repository` .

Nous en avons presque terminé avec cette section. Maintenant que vous avez converti le JSON en objets C#, nous allons afficher le nom de chaque dépôt. Remplacez les lignes :

```
var msg = await stringTask;   //**Deleted this
Console.Write(msg);
```

avec les éléments suivants :

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

Compilez et exécutez l'application. Cela affichera les noms des espaces de stockage qui font partie de la .NET Foundation.

Contrôle de la sérialisation

Avant d'ajouter d'autres fonctionnalités, nous allons traiter la propriété à l' `name` aide de l' `[JsonPropertyName]` attribut. Apportez les modifications suivantes à la déclaration du champ `name` dans repo.cs :

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

Pour utiliser `[JsonPropertyName]` l'attribut, vous devez ajouter l' System.Text.Json.Serialization espace de noms aux `using` directives :

```
using System.Text.Json.Serialization;
```

Cette modification signifie que vous devez modifier le code qui écrit le nom de chaque dépôt dans program.cs :

```
Console.WriteLine(repo.Name);
```

Exécutez `dotnet run` pour vérifier que les mappages sont corrects. Vous devriez voir la même sortie que précédemment.

Nous allons apporter une modification de plus avant d'ajouter de nouvelles fonctionnalités. La méthode `ProcessRepositories` peut faire le travail de façon asynchrone et renvoyer une collection des espaces de stockage. Nous allons renvoyer `List<Repository>` à partir de cette méthode et déplacer le code qui écrit les informations dans la méthode `Main`.

Modifiez la signature de `ProcessRepositories` pour retourner une tâche dont le résultat est une liste d'objets `Repository` :

```
private static async Task<List<Repository>> ProcessRepositories()
```

Ensuite, retournez simplement les dépôts après le traitement de la réponse JSON :

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

Le compilateur génère l'objet `Task<T>` pour la valeur de retour, car vous avez marqué cette méthode comme `async`. Ensuite, nous allons modifier la méthode `Main` afin qu'elle capture les résultats et écrive le nom de chaque dépôt dans la console. Votre méthode `Main` se présente maintenant comme suit :

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

Lire plus d'informations

Terminons par le traitement d'un certain nombre de propriétés dans le paquet JSON envoyé à partir de l'API GitHub. Vous ne voudrez pas récupérer tous les éléments, mais ajouter quelques propriétés vous permettra de découvrir d'autres fonctionnalités du langage C#.

Commençons par ajouter quelques autres types simples à la définition de classe `Repository`. Ajoutez ces propriétés à cette classe :

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

Ces propriétés ont intégré des conversions intégrées du type chaîne (c'est ce que contiennent les paquets JSON) vers le type cible. Le type Uri peut être nouveau pour vous. Il représente un URI, ou dans ce cas, une URL. Dans le cas des types `Uri` et `int`, si le paquet JSON contient des données qui ne sont pas convertibles dans le type cible, l'action de sérialisation lèvera une exception.

Une fois que vous avez ajouté cela, mettez à jour la méthode `Main` pour afficher ces éléments :

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

Dans la dernière étape, nous allons ajouter les informations de la dernière opération Push. Ces informations sont formatées de cette façon dans la réponse JSON :

```
2016-02-08T21:27:00Z
```

Ce format est au format de temps universel coordonné (UTC). vous obtiendrez donc une DateTime valeur dont la Kind propriété est Utc . Si vous préférez une date représentée dans votre fuseau horaire, vous devez écrire une méthode de conversion personnalisée. Tout d'abord, définissez une `public` propriété qui contiendra la représentation UTC de la date et de l'heure dans votre `Repository` classe et une `LastPush` `readonly` propriété qui retourne la date convertie en heure locale :

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

Passons en revue les nouvelles constructions que nous venons de définir. La `LastPush` propriété est définie à l'aide d'un *membre à expression nulle* pour l' `get` accesseur. Il n'existe aucun accesseur `set`. L'omission de l' `set` accesseur est la façon dont vous définissez une propriété *en lecture seule* en C#. (Oui, vous pouvez créer des propriétés *en écriture seule* en C#, mais leur intérêt est limité.)

Enfin, ajoutez une dernière instruction de sortie dans la console et vous êtes prêt à générer et exécuter de nouveau cette application :

```
Console.WriteLine(repo.LastPush);
```

Votre version doit maintenant correspondre à l'exemple terminé.

Conclusion

Ce didacticiel vous a montré comment effectuer des demandes web, analyser le résultat et afficher les propriétés de ces résultats. Vous avez également ajouté de nouveaux packages en tant que dépendances dans votre projet. Vous avez vu certaines des fonctionnalités du langage C# qui prennent en charge des techniques orientées objet.

Vous n'avez pas à exécuter `dotnet restore`, car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration, comme,,,, `dotnet new` `dotnet build` `dotnet run` `dotnet test` `dotnet publish` et `dotnet pack`. Pour désactiver la restauration implicite, utilisez l' `--no-restore` option.

La `dotnet restore` commande est toujours utile dans certains scénarios où la restauration explicite est appropriée, comme les Builds d'intégration continue dans Azure DevOps services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

Pour plus d'informations sur la gestion des flux NuGet, consultez la `dotnet restore` documentation.

Héritage dans C# et .NET

02/11/2020 • 46 minutes to read • Edit Online

Ce didacticiel vous présente l'héritage dans C#. L'héritage est une fonctionnalité des langages de programmation orientés objet qui vous permet de définir une classe de base qui fournit des fonctionnalités spécifiques (données et comportement) et de définir des classes dérivées qui héritent ou substituent cette fonctionnalité.

Prérequis

Ce didacticiel part du principe que vous avez installé le kit SDK .NET Core. Visitez la page des téléchargements .net Core pour le télécharger. Il vous faut également un éditeur de code. Ce didacticiel utilise Visual Studio Code, mais vous pouvez utiliser l'éditeur de code de votre choix.

Exécution des exemples

Pour créer et exécuter les exemples de ce didacticiel, vous utilisez l'utilitaire dotnet en ligne de commande. Pour chaque exemple, procédez comme suit :

1. Créez un répertoire pour stocker l'exemple.
2. Entrez la commande dotnet new console dans l'invite de commandes pour créer un projet .NET Core.
3. Copiez et collez le code de l'exemple dans votre éditeur de code.
4. Entrez la commande dotnet restore à partir de la ligne de commande pour charger ou restaurer les dépendances du projet.

   Vous n'avez pas à exécuter `dotnet restore`, car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration, comme,,,, `dotnet new` `dotnet build` `dotnet run` `dotnet test` `dotnet publish` et `dotnet pack`. Pour désactiver la restauration implicite, utilisez l' `--no-restore` option.

   La `dotnet restore` commande est toujours utile dans certains scénarios où la restauration explicite est appropriée, comme les Builds d'intégration continue dans Azure DevOps services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

   Pour plus d'informations sur la gestion des flux NuGet, consultez la `dotnet restore` documentation.
5. Entrez la commande dotnet run pour compiler et exécuter l'exemple.

Présentation : Qu'est-ce que l'héritage ?

L'héritage est un des attributs fondamentaux de la programmation orientée objet. Il vous permet de définir une classe enfant qui réutilise (hérite), étend ou modifie le comportement d'une classe parente. La classe dont les membres sont hérités s'appelle la *classe de base*. La classe qui hérite des membres de la classe de base est appelée la *classe dérivée*.

C# et .NET prennent uniquement en charge *l'héritage simple*. C'est-à-dire qu'une classe ne peut hériter que d'une seule classe. Toutefois, l'héritage est transitif, ce qui permet de définir une hiérarchie d'héritage pour un ensemble de types. En d'autres termes, le type `D` peut hériter du type `C`, qui hérite du type `B`, qui hérite du type de classe de base `A`. Étant donné que l'héritage est transitif, les membres de type `A` sont disponibles pour le type `D`.

Tous les membres d'une classe de base ne sont pas hérités par les classes dérivées. Les membres suivants ne sont pas hérités :

- Les constructeurs statiques, qui initialisent les données statiques d'une classe.
- Les Constructeurs d'instance, que vous appelez pour créer une nouvelle instance de la classe. Chaque classe doit définir ses propres constructeurs.
- Les finaliseurs, qui sont appelés par le récupérateur de mémoire du runtime pour détruire les instances d'une classe.

Bien que tous les autres membres de classe de base sont hérités par les classes dérivées, leur visibilité dépend de leur accessibilité. L'accessibilité d'un membre affecte sa visibilité pour les classes dérivées de la manière suivante :

- Les membres Privés sont visibles uniquement dans les classes dérivées qui sont imbriquées dans leur classe de base. Sinon, ils ne sont pas visibles dans les classes dérivées. Dans l'exemple suivant, `A.B` est une classe imbriquée qui dérive de `A`, et `C` dérive de `A`. Le champ privé `A.value` est visible dans A.B. Toutefois, si vous supprimez les commentaires de la méthode `C.GetValue` et essayez de compiler l'exemple, il génère l'erreur de compilateur CS0122 : « 'A.value est inaccessible en raison de son niveau de protection ».

```
using System;

public class A
{
   private int value = 10;

   public class B : A
   {
       public int GetValue()
       {
           return this.value;
       }
   }
}

public class C : A
{
//    public int GetValue()
//    {
//        return this.value;
//    }
}

public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- Les membres protégés sont visibles uniquement dans les classes dérivées.
- Les membres internes sont visibles uniquement dans les classes dérivées qui sont trouvent dans le même assembly que la classe de base. Ils ne sont pas visibles dans les classes dérivées situées dans un autre assembly à partir de la classe de base.
- Les membres publics sont visibles dans les classes dérivées et font partie de l'interface publique de la classe dérivée. Les membres publics hérités peuvent être appelés comme s'ils étaient définis dans la classe dérivée. Dans l'exemple suivant, la classe `A` définit une méthode nommée `Method1`, et la classe `B` hérite de la classe `A`. L'exemple appelle ensuite `Method1` comme s'il s'agissait d'une méthode d'instance sur `B`.

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}
```

Les classes dérivées peuvent également *substituer* les membres hérités en fournissant une implémentation alternative. Pour être en mesure de substituer un membre, le membre de la classe de base doit être marqué avec le mot-clé virtual. Par défaut, les membres de classe de base ne sont pas marqués comme `virtual` et ne peut pas être substitués. Une tentative de substituer un membre non virtuel, comme dans l'exemple suivant, génère l'erreur de compilateur CS0506 : « <member> : impossible de substituer le membre hérité <member>, car il n'est pas marqué comme virtual, abstract ou override.

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

Dans certains cas, une classe dérivée *doit* remplacer l'implémentation de la classe de base. Les membres de classe de base marqués avec le mot-clé abstract requièrent que les classes dérivées les remplacent. Compiler l'exemple suivant génère l'erreur de compilateur CS0534, « <class> does not implement inherited abstract member <member> », car la classe `B` ne fournit aucune implémentation de `A.Method1`.

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

L'héritage s'applique uniquement aux classes et interfaces. Les autres catégories de type (structures, délégués et énumérations) ne permettent pas l'héritage. En raison de ces règles, la tentative de compilation de code comme l'exemple suivant génère l'erreur de compilateur CS0527 : « Le type 'ValueType' dans la liste des interfaces n'est pas une interface ». Le message d'erreur indique que, même si vous pouvez définir les interfaces qu'implémente un struct, l'héritage n'est pas pris en charge.

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Héritage implicite

Outre les types qui peuvent hériter via l'héritage simple, tous les types dans le système de types de .NET héritent implicitement de Object ou d'un type dérivé. Les fonctionnalités communes de Object sont disponibles pour n'importe quel type.

Pour comprendre ce que l'héritage implicite signifie, nous allons définir une nouvelle classe, `SimpleClass`, qui est simplement une définition de classe vide :

```
public class SimpleClass
{ }
```

Vous pouvez ensuite utiliser la réflexion (qui permet d'inspecter les métadonnées d'un type pour obtenir des informations sur ce type) pour obtenir la liste des membres qui appartiennent au type `SimpleClass`. Même si vous n'avez pas défini de membres dans votre classe `SimpleClass`, la sortie de l'exemple indique qu'il a en fait neuf membres. Un de ces membres est un constructeur sans paramètre (ou par défaut) qui est fourni automatiquement pour le type `SimpleClass` par le compilateur C#. Les huit restants sont membres de Object, le type à partir duquel toutes les classes et interfaces du système de type .NET héritent implicitement.

```
using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
                             BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by
{member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method):  Public, Declared by System.Object
// Equals (Method):  Public, Declared by System.Object
// Equals (Method):  Public Static, Declared by System.Object
// ReferenceEquals (Method):  Public Static, Declared by System.Object
// GetHashCode (Method):  Public, Declared by System.Object
// GetType (Method):  Public, Declared by System.Object
// Finalize (Method):  Internal, Declared by System.Object
// MemberwiseClone (Method):  Internal, Declared by System.Object
// .ctor (Constructor):  Public, Declared by SimpleClass
```

L'héritage implicite à partir de la classe Object rend ces méthodes disponibles pour la classe `SimpleClass` :

- La méthode public `ToString`, qui convertit un objet `SimpleClass` en sa représentation de chaîne, retourne le nom de type complet. Dans ce cas, la méthode `ToString` retourne la chaîne « SimpleClass ».
- Voici trois méthodes de test d'égalité de deux objets : la méthode public instance `Equals(Object)`, la méthode public static `Equals(Object, Object)` et la méthode public static `ReferenceEquals(Object, Object)`. Par défaut, ces méthodes testent l'égalité des références. Autrement dit, pour être égales, deux variables d'objet doivent faire référence au même objet.
- La méthode public `GetHashCode`, qui calcule une valeur qui permet à une instance du type d'être utilisée dans des collections hachées.
- La méthode `GetType` publique qui retourne un objet Type qui représente le type `SimpleClass`.

- La méthode protected Finalize, qui est conçue pour libérer les ressources non gérées avant que la mémoire d'un objet soit récupérée par le récupérateur de mémoire.
- La méthode protected MemberwiseClone, qui crée un clone partiel de l'objet actuel.

En raison de l'héritage implicite, vous pouvez appeler n'importe quel membre hérité d'un objet `SimpleClass` exactement comme s'il était en fait un membre défini dans la classe `SimpleClass`. Par exemple, l'exemple suivant appelle la méthode `SimpleClass.ToString`, dont `SimpleClass` hérite de Object.

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        SimpleClass
```

Le tableau suivant répertorie les catégories de types que vous pouvez créer en C# et les types à partir desquels ils héritent implicitement. Chaque type de base apporte un autre ensemble de membres disponibles via l'héritage aux types dérivés implicitement.

CATÉGORIE DE TYPE	HÉRITE IMPLICITEMENT DE
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
délégué	MulticastDelegate, Delegate, Object

L'héritage et une relation « est un »

En règle générale, l'héritage est utilisé pour exprimer une relation « est un » entre une classe de base et une ou plusieurs classes dérivées, où les classes dérivées sont des versions spécialisées de la classe de base ; la classe dérivée est un type de la classe de base. Par exemple, la classe `Publication` représente une publication de tout type et les classes `Book` et `Magazine` représentent les classes des types spécifiques de publications.

> **NOTE**
> Les classes et structures peuvent implémenter une ou plusieurs interfaces. Bien que l'implémentation d'interface est souvent présentée comme une solution de contournement pour l'héritage unique ou comme une façon d'utiliser l'héritage avec les structures, elle est conçue pour exprimer une autre relation (« peut faire ») entre une interface et son type d'implémentation que l'héritage. Une interface définit un sous-ensemble de fonctionnalités (comme la capacité à tester l'égalité, comparer ou trier des objets, ou pour prendre en charge la mise en forme et l'analyse dépendant de la culture) que l'interface met à disposition pour ses types d'implémentation.

Notez que « est un » exprime également la relation entre un type et une instanciation spécifique de ce type. Dans

l'exemple suivant, `Automobile` est une classe qui possède trois propriétés en lecture seule uniques : `Make`, le fabricant de l'automobile ; `Model`, le type de voiture et `Year`, son année de fabrication. Votre classe `Automobile` comporte également un constructeur dont les arguments sont assignés aux valeurs de propriété, et elle remplace la méthode Object.ToString pour générer une chaîne qui identifie de façon unique l'instance `Automobile` plutôt que la classe `Automobile`.

```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
           throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrWhiteSpace(make))
           throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
           throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrWhiteSpace(model))
           throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
           throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

Dans ce cas, vous ne devriez pas vous reposer sur l'héritage pour représenter les modèles et constructeurs spécifiques. Par exemple, il est inutile de définir un type `Packard` pour représenter les véhicules automobiles fabriqués par la société Packard Motor Car. Au lieu de cela, vous pouvez les représenter en créant un objet `Automobile` avec les valeurs appropriées passées à son constructeur de classe, comme dans l'exemple suivant.

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//        1948 Packard Custom Eight
```

Une relation « est un » basée sur l'héritage est préférablement appliquée à une classe de base et aux classes dérivées qui ajoutent des membres supplémentaires à la classe de base ou qui nécessitent des fonctionnalités supplémentaires non présentes dans la classe de base.

Conception de la classe de base et des classes dérivées

Examinons le processus de conception d'une classe de base et de ses classes dérivées. Dans cette section, vous allez définir une classe de base, `Publication` , qui représente une publication de tout type, comme un livre, un magazine, un journal, un journal, un article, etc. Vous définirez également une `Book` classe qui dérive de `Publication` . Vous pourriez facilement étendre l'exemple pour définir d'autres classes dérivées, comme `Magazine`, `Journal`, `Newspaper` et `Article`.

Classe Publication de base

Lors de la conception de votre classe `Publication`, vous devez prendre plusieurs décisions de conception :

- Les membres à inclure dans votre classe de base `Publication` et si les membres `Publication` fournissent des implémentations de méthode, ou si `Publication` est une classe de base abstraite qui sert de modèle pour ses classes dérivées.

  Dans ce cas, la classe `Publication` fournit des implémentations de méthode. La section Conception de classes de base abstraites et leurs classes dérivées contient un exemple qui utilise une classe de base abstraite pour définir les méthodes que les classes dérivées doivent substituer. Les classes dérivées sont libres de fournir une implémentation qui convient pour le type dérivé.

  La possibilité de réutiliser le code (autrement dit, plusieurs classes dérivées partagent la déclaration et l'implémentation de méthodes de classe de base et n'ont pas besoin de les substituer) constitue un avantage des classes de base non abstraites. Par conséquent, vous devez ajouter des membres à `Publication` si leur code est susceptible d'être partagé par certains ou la majorité des types `Publication` spécialisés. Si vous ne fournissez pas efficacement les implémentations de classe de base, vous devrez fournir des implémentations de membres en grande partie identiques dans les classes dérivées au lieu d'une implémentation unique dans la classe de base. La nécessité de maintenir le code dupliqué à plusieurs emplacements est une source potentielle de bogues.

  Pour optimiser la réutilisation du code et créer une hiérarchie d'héritage logique et intuitive, vous voulez vous assurer d'ajouter à la classe `Publication` uniquement les données et fonctionnalités communes à toutes les publications ou la plupart d'entre elles. Les classes dérivées implémentent ensuite les membres qui sont uniques pour les types particuliers de publications qu'ils représentent.

- La mesure dans laquelle étendre votre hiérarchie de classes. Souhaitez-vous développer une hiérarchie de trois classes ou plus, plutôt que simplement une classe de base et une ou plusieurs classes dérivées ? Par exemple, `Publication` peut être une classe de base de `Periodical`, qui est elle-même une classe de base de `Magazine`, `Journal` et `Newspaper`.

  Dans votre exemple, vous allez utiliser la hiérarchie simple d'une classe `Publication` et d'une classe dérivée unique, `Book`. Vous pouvez facilement étendre l'exemple pour créer un certain nombre de classes supplémentaires qui dérivent de `Publication` , telles que `Magazine` et `Article` .

- S'il est judicieux d'instancier la classe de base. Si ce n'est pas le cas, vous devez appliquer le mot-clé abstract à la classe. Dans le cas contraire, votre classe `Publication` peut être instanciée en appelant son constructeur de classe. Si une tentative est faite pour instancier une classe marquée avec le mot-clé `abstract` par un appel direct à son constructeur de classe, le compilateur C# génère l'erreur CS0144, « Impossible de créer une instance de la classe ou interface abstraite ». Si une tentative est faite pour instancier la classe à l'aide de la réflexion, la méthode de la réflexion lève une exception MemberAccessException.

  Par défaut, une classe de base peut être instanciée en appelant son constructeur de classe. Vous n'avez pas à définir explicitement un constructeur de classe. S'il n'y en a aucun présent dans le code source de la classe de base, le compilateur C# fournit automatiquement un constructeur par défaut (sans paramètre).

  Dans votre exemple, vous marquez la classe `Publication` comme abstract afin qu'elle ne puisse pas être

instanciée. Une classe `abstract` sans méthode `abstract` indique que cette classe représente un concept abstrait qui est partagé entre plusieurs classes concrètes (comme `Book`, `Journal`).

- Si les classes dérivées doivent hériter de l'implémentation de classe de base de membres particuliers, si elles ont l'option de substituer l'implémentation de la classe de base ou si elles doivent fournir une implémentation. Vous utilisez le mot clé abstract pour forcer les classes dérivées à fournir une implémentation. Vous devez utiliser le mot clé virtual pour permettre aux classes dérivées de substituer une méthode de classe de base. Par défaut, les méthodes définies dans la classe de base ne sont *pas* substituables.

  La classe `Publication` n'a aucune méthode `abstract`, mais la classe elle-même est `abstract`.

- Si une classe dérivée représente la classe finale dans la hiérarchie d'héritage et ne peut pas elle-même être utilisée comme classe de base pour les classes dérivées supplémentaires. Par défaut, toute classe peut servir de classe de base. Vous pouvez appliquer le mot clé sealed pour indiquer qu'une classe ne peut pas servir de classe de base pour des classes supplémentaires. La tentative de dériver à partir d'une classe sealed a généré l'erreur de compilateur CS0509, « impossible de dériver à partir du type sealed <typeName> ».

  Dans votre exemple, vous allez marquer votre classe dérivée en tant que `sealed`.

L'exemple suivant montre le code source pour la classe `Publication` ainsi qu'une énumération `PublicationType` retournée par la propriété `Publication.PublicationType`. Outre les membres qu'elle hérite de Object, la classe `Publication` définit les membres uniques et substitutions de membres suivants :

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
   private bool published = false;
   private DateTime datePublished;
   private int totalPages;

   public Publication(string title, string publisher, PublicationType type)
   {
      if (String.IsNullOrWhiteSpace(publisher))
         throw new ArgumentException("The publisher is required.");
      Publisher = publisher;

      if (String.IsNullOrWhiteSpace(title))
         throw new ArgumentException("The title is required.");
      Title = title;

      Type = type;
   }

   public string Publisher { get; }

   public string Title { get; }

   public PublicationType Type { get; }

   public string CopyrightName { get; private set; }

   public int CopyrightDate { get; private set; }

   public int Pages
   {
     get { return totalPages; }
     set
     {
         if (value <= 0)
```

```
            throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
         totalPages = value;
      }
   }

   public string GetPublicationDate()
   {
      if (!published)
         return "NYP";
      else
         return datePublished.ToString("d");
   }

   public void Publish(DateTime datePublished)
   {
      published = true;
      this.datePublished = datePublished;
   }

   public void Copyright(string copyrightName, int copyrightDate)
   {
      if (String.IsNullOrWhiteSpace(copyrightName))
         throw new ArgumentException("The name of the copyright holder is required.");
      CopyrightName = copyrightName;

      int currentYear = DateTime.Now.Year;
      if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
         throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and
{currentYear + 1}");
      CopyrightDate = copyrightDate;
   }

   public override string ToString() => Title;
}
```

- Un constructeur

  Étant donné que la classe `Publication` est `abstract`, elle ne peut pas être instanciée directement à partir du code comme dans l'exemple suivant :

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                    PublicationType.Book);
  ```

  Toutefois, son constructeur d'instance peut être appelé directement à partir des constructeurs de classes dérivées, comme le code source pour la classe `Book` le montre.

- Deux propriétés liées à la publication

  `Title` est une propriété String en lecture seule dont la valeur est fournie en appelant le constructeur `Publication`.

  `Pages` est une propriété Int32 en lecture seule qui indique le nombre total de pages de la publication. La valeur est stockée dans un champ privé nommé `totalPages`. Elle doit être positive, sans quoi une exception ArgumentOutOfRangeException est levée.

- Membres liés à l'éditeur

  Deux propriétés en lecture seule, `Publisher` et `Type`. Les valeurs sont à l'origine fournies par l'appel au constructeur de la classe `Publication`.

- Membres liés à la publication

  Deux méthodes, `Publish` et `GetPublicationDate` définissent et retournent la date de publication. La

méthode `Publish` définit l'indicateur privé `published` sur `true` lorsqu'elle est appelée et affecte la date passée comme argument au champ privé `datePublished`. La méthode `GetPublicationDate` retourne la chaîne « NYP » si l'indicateur `published` est `false`, et la valeur du champ `datePublished` si l'indicateur est `true`.

- Membres liés aux droits d'auteur

  La méthode `Copyright` prend comme arguments le nom du titulaire des droits d'auteur et l'année des droits d'auteur et les attribue aux propriétés `CopyrightName` et `CopyrightDate`.

- Une substitution de la méthode `ToString`

  Si un type ne remplace pas la méthode Object.ToString, il retourne le nom qualifié complet du type, ce qui n'aide pas vraiment à faire la différence entre une instance et une autre. La classe `Publication` substitue Object.ToString pour retourner la valeur de la propriété `Title`.

Le schéma suivant illustre la relation entre votre classe `Publication` de base et sa classe Object implicitement héritée.

La classe `Book`

La classe `Book` représente un livre sous la forme d'un type spécialisé de publication. L'exemple suivant montre le code source pour la classe `Book`.

```
using System;

public sealed class Book : Publication
{
   public Book(string title, string author, string publisher) :
          this(title, String.Empty, author, publisher)
   { }

   public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
   {
      // isbn argument must be a 10- or 13-character numeric string without "-" characters.
      // We could also determine whether the ISBN is valid by comparing its checksum digit
      // with a computed checksum.
      //
      if (! String.IsNullOrEmpty(isbn)) {
        // Determine if ISBN length is correct.
        if (! (isbn.Length == 10 | isbn.Length == 13))
            throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
        ulong nISBN = 0;
        if (! UInt64.TryParse(isbn, out nISBN))
            throw new ArgumentException("The ISBN can consist of numeric characters only.");
      }
      ISBN = isbn;

      Author = author;
   }

   public string ISBN { get; }

   public string Author { get; }

   public Decimal Price { get; private set; }

   // A three-digit ISO currency symbol.
   public string Currency { get; private set; }

   // Returns the old price, and sets a new price.
   public Decimal SetPrice(Decimal price, string currency)
   {
       if (price < 0)
          throw new ArgumentOutOfRangeException("The price cannot be negative.");
       Decimal oldValue = Price;
       Price = price;

       if (currency.Length != 3)
          throw new ArgumentException("The ISO currency symbol is a 3-character string.");
       Currency = currency;

       return oldValue;
   }

   public override bool Equals(object obj)
   {
      Book book = obj as Book;
      if (book == null)
         return false;
      else
         return ISBN == book.ISBN;
   }

   public override int GetHashCode() => ISBN.GetHashCode();

   public override string ToString() => $"{(String.IsNullOrEmpty(Author) ? "" : Author + ", ")}{Title}";
}
```

Outre les membres qu'elle hérite de `Publication`, la classe `Book` définit les membres uniques et substitutions de membres suivants :

- Deux constructeurs

  Les deux constructeurs `Book` partagent trois paramètres communs. Deux d'entre eux, *title* et *publisher*, correspondent aux paramètres du constructeur `Publication`. Le troisième est *author*, qui est stocké dans une propriété `Author` non modifiable publique. Un constructeur inclut un paramètre *isbn*, qui est stocké dans l'auto-propriété `ISBN`.

  Le premier constructeur utilise le mot-clé this pour appeler l'autre constructeur. Le chaînage de constructeurs est un modèle courant pour la définition de constructeurs. Les constructeurs avec le moins de paramètres fournissent les valeurs par défaut au moment de l'appel du constructeur avec le plus grand nombre de paramètres.

  Le deuxième constructeur utilise le mot-clé base pour transmettre le titre et le nom de l'éditeur au constructeur de classe de base. Si vous n'effectuez pas un appel explicite à un constructeur de classe de base dans votre code source, le compilateur C# fournit automatiquement un appel au constructeur par défaut ou sans paramètres de la classe de base.

- Une propriété `ISBN` en lecture seule qui retourne le numéro ISBN de l'objet `Book`, un numéro unique à 10 ou 13 chiffres. Le numéro ISBN est fourni en tant qu'argument à un des constructeurs `Book`. Le numéro ISBN est stocké dans un champ de stockage privé qui est généré automatiquement par le compilateur.

- Une propriété `Author` en lecture seule. Le nom de l'auteur est fourni en tant qu'argument aux deux constructeurs `Book` et est stocké dans la propriété.

- Deux propriétés en lecture seule relatives au prix, `Price` et `Currency`. Leurs valeurs sont fournies comme arguments dans un appel de méthode `SetPrice`. La propriété `Currency` est le symbole de devise ISO à trois caractères (par exemple, USD pour le dollar américain). Les symboles de devise ISO peuvent être récupérés à partir de la propriété ISOCurrencySymbol. Ces deux propriétés sont en lecture seule en externe, mais peuvent être définies par du code dans la classe `Book`.

- Une méthode `SetPrice` qui définit les valeurs des propriétés `Price` et `Currency`. Ces valeurs sont retournées par ces mêmes propriétés.

- Se substitue à la méthode `ToString` (héritée de `Publication`) et aux méthodes Object.Equals(Object) et GetHashCode (héritées de Object).

  Sauf si elle est substituée, la méthode Object.Equals(Object) teste l'égalité des références. Autrement dit, deux variables d'objet sont considérées comme égales si elles font référence au même objet. Dans la classe `Book`, en revanche, deux objets `Book` doivent être égaux s'ils ont le même ISBN.

  Lorsque vous substituez la méthode Object.Equals(Object), vous devez également substituer la méthode GetHashCode qui retourne une valeur que le runtime utilise pour stocker les éléments dans les collections hachées pour une récupération efficace. Le code de hachage doit retourner une valeur qui est cohérente avec le test d'égalité. Étant donné que vous avez substitué Object.Equals(Object) pour retourner `true` si les propriétés ISBN de deux objets `Book` sont égales, vous retournez le code de hachage calculé en appelant la méthode GetHashCode de la chaîne retournée par la propriété `ISBN`.

Le schéma suivant illustre la relation entre la base la classe `Book` et `Publication`, sa classe de base.

Publication
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Book
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

Key

Unique member	
Inherited member	
Overridden member	

Vous pouvez maintenant instancier un objet `Book`, appeler ses membres uniques et hérités le passer en tant qu'argument pour une méthode qui attend un paramètre de type `Publication` ou de type `Book`, comme illustré dans l'exemple suivant.

```
using System;
using static System.Console;

public class Example
{
   public static void Main()
   {
      var book = new Book("The Tempest",  "0971655819", "Shakespeare, William",
                          "Public Domain Press");
      ShowPublicationInfo(book);
      book.Publish(new DateTime(2016, 8, 18));
      ShowPublicationInfo(book);

      var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
      Write($"{book.Title} and {book2.Title} are the same publication: " +
            $"{((Publication) book).Equals(book2)}");
   }

   public static void ShowPublicationInfo(Publication pub)
   {
       string pubDate = pub.GetPublicationDate();
       WriteLine($"{pub.Title}, " +
                 $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}");
   }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Conception de classes de base abstraites et de leurs classes dérivées

Dans l'exemple précédent, vous avez défini une classe de base qui a fourni une implémentation d'un certain nombre de méthodes pour permettre aux classes dérivées de partager du code. Dans de nombreux cas, toutefois, la classe de base n'est pas censée fournir une implémentation. Au lieu de cela, la classe de base est une *classe abstraite* qui déclare des *méthodes abstraites* ; elle sert de modèle qui définit les membres que chaque classe dérivée doit implémenter. En général, dans une classe de base abstraite, l'implémentation de chaque type dérivé est unique pour ce type. Vous avez marqué la classe avec le mot clé abstract, car il n'était pas judicieux d'instancier un objet `Publication`, même si la classe fournissait des implémentations de fonctionnalités communes aux publications.

Par exemple, chaque forme géométrique bidimensionnelle fermée inclut deux propriétés : l'aire, l'étendue interne de la forme, et le périmètre, ou la distance le long des bords de la forme. La façon de laquelle ces propriétés sont calculées, cependant, dépend entièrement de la forme spécifique. La formule pour calculer le périmètre (ou la circonférence) d'un cercle, par exemple, est différente de celle d'un triangle. La classe `Shape` est une classe `abstract` avec des méthodes `abstract`. Ceci indique que les classes dérivées partagent les mêmes fonctionnalités, mais ces classes dérivées implémentent ces fonctionnalités différemment.

L'exemple suivant définit une classe de base abstraite nommée `Shape` qui définit deux propriétés : `Area` et `Perimeter`. En plus de marquer la classe avec le mot clé abstract, chaque membre de l'instance est également marqué avec le mot clé abstract. Dans ce cas, `Shape` substitue également la méthode Object.ToString pour renvoyer le nom du type, plutôt que son nom qualifié complet. Elle définit aussi deux membres statiques, `GetArea` et `GetPerimeter`, qui permettent aux appelants de récupérer facilement l'aire et le périmètre d'une instance de toute classe dérivée. Lorsque vous passez une instance d'une classe dérivée à l'une de ces méthodes, le runtime appelle la substitution de la méthode de la classe dérivée.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }

   public abstract double Perimeter { get; }

   public override string ToString() => GetType().Name;

   public static double GetArea(Shape shape) => shape.Area;

   public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Vous pouvez ensuite dériver des classes qui représentent des formes spécifiques à partir de `Shape`. L'exemple suivant définit trois classes : `Triangle`, `Rectangle` et `Circle`. Chaque forme utilise une formule unique pour calculer l'aire et périmètre. Certaines des classes dérivées définissent également des propriétés, telles que `Rectangle.Diagonal` et `Circle.Diameter`, qui sont propres à la forme qu'ils représentent.

```
using System;

public class Square : Shape
{
   public Square(double length)
   {
      Side = length;
   }

   public double Side { get; }

   public override double Area => Math.Pow(Side, 2);

   public override double Perimeter => Side * 4;

   public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; }

   public double Width { get; }

   public override double Area => Length * Width;

   public override double Perimeter => 2 * Length + 2 * Width;

   public bool IsSquare() => Length == Width;

   public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

   public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

   // Define a circumference, since it's the more familiar term.
   public double Circumference => Perimeter;

   public double Radius { get; }

   public double Diameter => Radius * 2;
}
```

L'exemple suivant utilise les objets dérivés de `Shape`. Elle instancie un tableau d'objets dérivés de `Shape` et appelle les méthodes statiques de la classe `Shape` qui encapsule les valeurs de propriété de retour de `Shape`. Le runtime récupère les valeurs de propriétés substituées des types dérivés. L'exemple convertit également chaque objet `Shape` dans le tableau en son type dérivé et, si la conversion réussit, récupère les propriétés de cette sous-classe particulière de `Shape`.

```
using System;

public class Example
{
   public static void Main()
   {
      Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                         new Circle(3) };
      foreach (var shape in shapes) {
         Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                           $"perimeter, {Shape.GetPerimeter(shape)}");
         var rect = shape as Rectangle;
         if (rect != null) {
            Console.WriteLine($"   Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
            continue;
         }
         var sq = shape as Square;
         if (sq != null) {
            Console.WriteLine($"   Diagonal: {sq.Diagonal}");
            continue;
         }
      }
   }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Voir aussi

- Héritage (Guide de programmation C#)

Utiliser LINQ (Language-Integrated Query)

02/11/2020 • 28 minutes to read • Edit Online

Introduction

Ce tutoriel vous présente les fonctionnalités de .NET Core et du langage C#. Vous découvrirez comment effectuer les actions suivantes :

- Générer des séquences avec LINQ.
- Méthodes d'écriture qui peuvent être facilement utilisées dans les requêtes LINQ.
- Distinguer les évaluations hâtif et tardive.

Vous apprendrez ces techniques en créant une application qui illustre l'une des compétences de base de tout magicien : le mélange faro. En quelques mots, le mélange faro est une technique qui consiste à diviser un paquet de cartes en deux moitiés exactes, puis à intercaler une carte sur deux de chacune des deux moitiés de façon à reconstruire le jeu d'origine.

Les magiciens utilisent cette technique parce que chaque carte se trouve à un emplacement connu après chaque mélange, suivant un motif répétitif.

Dans notre cas, c'est une façon plaisante d'envisager la manipulation de séquences de données. L'application que vous allez générer construit un jeu de cartes, puis effectue une séquence de mélange, en écrivant chaque séquence à chaque fois. Vous comparerez également le nouvel ordre à l'ordre d'origine.

Ce didacticiel comporte plusieurs étapes. Après chaque étape, vous pourrez exécuter l'application et voir la progression. Vous pouvez également voir l'exemple terminé dans le dépôt GitHub dotnet/samples. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET core. Vous trouverez les instructions d'installation sur la page de téléchargement de .net Core . Vous pouvez exécuter cette application sur Windows, Ubuntu Linux ou OS X, ou dans un conteneur d'ancrage. Vous devez installer l'éditeur de code de votre choix. Les descriptions ci-dessous utilisent Visual Studio code qui est un éditeur multiplateforme Open source. Cependant, vous pouvez utiliser les outils avec lesquels vous êtes le plus à l'aise.

Création de l'application

La première étape consiste à créer une nouvelle application. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel. Saisissez la commande `dotnet new console` à l'invite. Elle crée les fichiers de démarrage d'une application « Hello World » de base.

Si vous n'avez jamais utilisé C#, ce didacticiel explique la structure d'un programme C#. Vous pouvez le lire, puis revenir ici pour en savoir plus sur LINQ.

Créer le jeu de données

Avant de commencer, vérifiez que les lignes suivantes figurent en haut du fichier `Program.cs` généré par `dotnet new console` :

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

Si ces trois lignes (instructions `using`) ne figurent pas en haut du fichier, le programme ne se compilera pas.

Maintenant que vous avez toutes les références dont vous avez besoin, réfléchissons à ce qui compose un jeu de cartes. En général, un jeu de cartes à jouer comporte quatre couleurs, et chaque couleur a treize valeurs. On pourrait envisager de créer directement une classe `Card` et de remplir manuellement une collection d'objets `Card`. Avec LINQ, il est possible d'être plus concis que cela. Au lieu de créer une classe `Card`, vous pouvez créer deux séquences pour représenter respectivement les couleurs et les rangs. Créez deux *méthodes d'itération* simples qui génèreront les rangs et les couleurs sous forme de IEnumerable<T> de chaînes :

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Placez-les sous la méthode `Main` dans votre fichier `Program.cs`. Les deux méthodes utilisent la syntaxe `yield return` pour produire une séquence lors de leur exécution. Le compilateur génère un objet qui implémente IEnumerable<T> et génère la séquence de chaînes au fur et à mesure.

Maintenant, utilisez ces méthodes d'itération pour créer le jeu de cartes. Placez la requête LINQ dans la méthode `Main` :

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

Les clauses `from` multiples produisent un SelectMany, qui crée une séquence unique en combinant chaque élément de la première séquence avec chaque élément de la deuxième séquence. L'ordre est important ici. Le premier élément de la première séquence source (couleurs) est associé à chacun des éléments de la deuxième séquence (rangs). Cette opération génère les treize cartes de la première couleur. Ce processus est reproduit pour chaque élément de la première séquence (couleurs). Le résultat final est un jeu de cartes classé par couleurs, puis par valeurs.

Il est important de garder à l'esprit la chose suivante : que vous choisissiez d'écrire votre code LINQ dans la syntaxe de requête utilisée plus haut ou d'utiliser plutôt la syntaxe de méthode, vous pourrez toujours passer d'une forme syntaxique à l'autre. La requête ci-dessus, écrite dans la syntaxe de requête, s'écrit ainsi dans la syntaxe de méthode :

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

Le compilateur traduit les instructions LINQ écrites avec la syntaxe de requête dans la syntaxe d'appel de méthode équivalente. Par conséquent, les deux versions de la requête produisent le même résultat quel que soit le choix de syntaxe. Choisissez la plus adaptée à votre situation : par exemple, si certains membres de votre équipe ont des difficultés à utiliser la syntaxe de méthode, privilégiez dans la mesure du possible la syntaxe de requête.

Ensuite, exécutez l'exemple que vous avez commencé à élaborer. Il affiche les 52 cartes du jeu. Il peut être très utile d'exécuter cet exemple avec un débogueur pour observer la façon dont les méthodes `Suits()` et `Ranks()` s'exécutent. Vous pouvez clairement voir que chaque chaîne de chaque séquence est générée uniquement au moment requis.

Manipuler la commande

Maintenant, réfléchissons à la façon dont nous allons battre les cartes du jeu. Pour bien faire, la première étape consiste à couper le jeu en deux. Les méthodes Take et Skip, qui font partie des API LINQ, offrent cette fonctionnalité. Placez-les sous la boucle `foreach` :

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

Cependant, il n'y a pas de méthode de battage dans la bibliothèque standard ; vous devez donc écrire la vôtre. Comme elle illustrera plusieurs techniques des programmes LINQ, nous allons expliquer les différentes parties du processus étape par étape.

Pour ajouter des fonctionnalités aux interactions possibles avec les IEnumerable<T> obtenus à partir des requêtes LINQ, vous allez écrire des méthodes d'un genre particulier, nommées méthodes d'extension. En bref, une méthode d'extension est une *méthode statique* spéciale qui ajoute de nouvelles fonctionnalités à un type existant sans qu'il soit nécessaire de modifier le type d'origine.

Pour accueillir vos méthodes d'extension, ajoutez à votre programme un nouveau fichier de classe *statique*, nommé `Extensions.cs`, puis créez la première méthode d'extension :

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T>
second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Examinez attentivement la signature de méthode, et en particulier les paramètres :

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

On peut voir l'ajout du modificateur `this` au premier argument de la méthode. Cela signifie que vous appelez la méthode comme s'il s'agissait d'une méthode membre du type du premier argument. Cette déclaration de méthode suit également un idiome standard selon lequel les types d'entrée et de sortie sont `IEnumerable<T>`. Cette pratique permet d'enchaîner les méthodes LINQ afin d'exécuter des requêtes plus complexes.

Bien entendu, le jeu ayant été coupé en deux, il faut réunir les deux moitiés. Dans le code, cela signifie que vous allez énumérer les séquences que vous avez acquises via Take et Skip à `interleaving` la fois, les éléments et la création d'une séquence : votre jeu de cartes à présent mélangé. Pour écrire une méthode LINQ qui fonctionne avec deux séquences, vous devez comprendre comment IEnumerable<T> fonctionne.

L'interface IEnumerable<T> possède une seule méthode : GetEnumerator. L'objet retourné par GetEnumerator a une méthode permettant d'atteindre l'élément suivant et une propriété qui récupère l'élément actif dans la séquence. Vous allez utiliser ces deux membres pour énumérer la collection et retourner les éléments. Cette méthode Interleave sera une méthode d'itération ; par conséquent, au lieu de créer une collection et de la retourner, vous allez utiliser la syntaxe `yield return` présentée ci-dessus.

Voici l'implémentation de cette méthode :

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Maintenant que vous avez écrit cette méthode, revenez à la méthode `Main` et mélangez le jeu une fois :

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Comparaisons

Combien de battages faut-il compter pour remettre le jeu dans l'ordre d'origine ? Pour le savoir, il faut écrire une méthode qui détermine si deux séquences sont égales. Ensuite, il vous faudra placer le code qui mélange le jeu dans une boucle et vérifier si le jeu est de nouveau dans l'ordre.

Vous ne devriez pas avoir de problèmes à écrire une méthode qui détermine si deux séquences sont égales. La structure est similaire à celle de la méthode que vous avez écrite pour mélanger le jeu. Seulement, cette fois, au lieu d'utiliser l'instruction `yield return` sur chaque élément, vous allez comparer les éléments correspondants de

chaque séquence. Une fois que la séquence aura été énumérée en entier, si tous les éléments correspondent, les séquences sont les mêmes :

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Cet exemple montre un autre terme LINQ : les méthodes terminales. Elles prennent une séquence en entrée (ou, dans ce cas, deux séquences) et retournent une valeur scalaire unique. Les méthodes terminales sont toujours la dernière méthode de la chaîne de méthodes d'une requête LINQ ; d'où le nom « terminal ».

Vous pourrez les voir en action lorsque vous les utiliserez pour déterminer si le jeu est dans l'ordre d'origine. Placez le code de mélange à l'intérieur d'une boucle, et arrêtez-la lorsque la séquence est à nouveau dans l'ordre d'origine en appliquant la méthode `SequenceEquals()`. Vous pouvez voir qu'il s'agit toujours de la dernière méthode d'une requête, parce qu'elle retourne une valeur unique plutôt qu'une séquence :

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Exécutez le code ainsi obtenu et étudiez la façon dont le jeu se réorganise à chaque battage. Après huit battages (itérations de la boucle do-while), le jeu revient à la configuration qu'il avait à sa création avec la requête LINQ initiale.

Optimisations

L'exemple que vous avez produit jusque-là exécute un *mélange extérieur*, où les cartes du haut et du bas restent les mêmes à chaque exécution. Faisons une modification en effectuant plutôt un *mélange intérieur*, où les 52 cartes changent toutes de position. Dans le cas d'un mélange intérieur, on intercale les deux moitiés du jeu de sorte que la première carte de la moitié du dessous devienne la première carte du jeu. Cela signifie que la dernière carte de la moitié du dessus devient la carte du bas. Cette modification simple ne concerne qu'une ligne de code. Mettez à jour la requête de battage actuelle en inversant les positions de Take et de Skip, de façon à modifier l'ordre des moitiés du dessus et du dessous du jeu :

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Réexécutez le programme : vous verrez qu'il faut 52 itérations pour que le jeu se réorganise. Vous commencerez également à remarquer de sérieuses dégradations des performances lors de l'exécution du programme.

Il y a plusieurs raisons à cela. L'une des causes principales de cette baisse de performances est une utilisation inefficace de *l'évaluation paresseuse*.

En bref, l'évaluation d'une instruction n'est effectuée que lorsque sa valeur devient nécessaire. Les requêtes LINQ sont évaluées de cette manière. Les séquences sont générées uniquement au moment où les éléments sont demandés. En règle générale, c'est un avantage majeur de LINQ. Toutefois, dans une utilisation du type de ce programme, cela entraîne une croissance exponentielle de la durée d'exécution.

N'oubliez pas que nous avons généré le jeu d'origine à l'aide d'une requête LINQ. Chaque mélange est généré en effectuant trois requêtes LINQ sur le jeu précédent. Toutes ces opérations sont effectuées de façon retardée. Cela signifie également qu'elles sont effectuées à chaque fois que la séquence est demandée. À la 52e itération, vous aurez régénéré le jeu d'origine de nombreuses fois. Nous allons écrire dans un journal pour illustrer ce comportement. Ensuite, vous résoudrez le problème.

Dans votre fichier `Extensions.cs`, tapez ou copiez la méthode d'extension ci-dessous. Elle crée un fichier nommé `debug.log` au sein de votre répertoire de projet et enregistre la requête en cours d'exécution dans le fichier journal. Elle peut être ajoutée à une requête pour marquer le fait qu'elle s'est exécutée.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Vous verrez une ligne ondulée rouge sous `File`, ce qui signifie qu'elle n'existe pas. Elle ne sera pas compilée, puisque le compilateur ne sait pas ce qu'est `File`. Pour résoudre ce problème, veillez à ajouter la ligne de code suivante sous la première ligne dans `Extensions.cs` :

```
using System.IO;
```

Cela devrait résoudre le problème et faire disparaître l'erreur rouge.

Instrumentez ensuite la définition de chaque requête avec un message de journalisation :

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Remarquez que vous n'écrivez pas dans le journal à chaque fois que vous accédez à une requête, mais seulement lors de la création de la requête d'origine. Le programme met toujours longtemps à s'exécuter, mais vous pouvez maintenant voir pourquoi. Si vous perdez patience au cours de l'exécution du mélange intérieur avec journalisation, revenez au mélange extérieur. Vous verrez quand même les effets de l'évaluation paresseuse. Sur une itération, elle exécute 2 592 requêtes, génération de toutes les valeurs et couleurs comprise.

Vous pouvez améliorer les performances du code en réduisant le nombre d'exécutions effectuées. L'un des correctifs les plus simples consiste à mettre en *cache* les résultats de la requête LINQ d'origine qui construit le jeu de cartes. Actuellement, les requêtes sont réexécutées chaque fois que la boucle do-while effectue une itération, construisant et battant de nouveau le jeu de cartes à chaque fois. Pour mettre en cache le jeu de cartes, vous pouvez utiliser les méthodes LINQ ToArray et ToList ; ajoutées aux requêtes, elles effectuent les actions voulues, mais en stockant respectivement les résultats dans un tableau ou dans une liste. Ajoutez la méthode LINQ ToArray aux deux requêtes et réexécutez le programme :

```
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Le mélange extérieur est descendu à 30 requêtes. Si vous repassez au mélange intérieur, vous constaterez des améliorations similaires : il ne comporte plus que 162 requêtes.

Sachez que cet exemple est **conçu** pour mettre en évidence les cas d'usage où l'évaluation paresseuse cause des problèmes de performances. S'il est important de voir son impact sur les performances du code, il faut également comprendre que toutes les requêtes ne doivent pas s'exécuter de manière stricte. La baisse de performances que l'on constate sans ToArray est due au fait que chaque nouvelle disposition du jeu de cartes est construite à partir de la configuration précédente. Avec l'évaluation paresseuse, chaque nouvelle configuration du jeu est générée à partir du jeu d'origine, y compris l'exécution du code qui a construit `startingDeck`, ce qui entraîne une grande quantité de travail supplémentaire.

Dans la pratique, certains algorithmes fonctionnent bien avec l'évaluation stricte, d'autres avec l'évaluation paresseuse. Au quotidien, l'évaluation paresseuse est en général un meilleur choix lorsque la source de données est un processus distinct, comme un moteur de base de données. L'évaluation paresseuse permet dans ce cas d'exécuter des requêtes plus complexes avec un seul aller-retour entre le processus de base de données et le reste du code. LINQ peut s'adapter tant à l'évaluation paresseuse qu'à l'évaluation stricte. Par conséquent, examinez vos

processus et choisissez le type d'évaluation qui vous offre les meilleures performances.

Conclusion

Dans ce projet, nous avons vu comment :

- utiliser des requêtes LINQ pour agréger des données en une séquence explicite ;
- écrire des méthodes d'extension pour ajouter des fonctionnalités personnalisées aux requêtes LINQ ;
- localiser les zones du code où les requêtes LINQ risquent de poser des problèmes de performances, comme une dégradation de la vitesse ;
- utiliser l'évaluation paresseuse et l'évaluation stricte dans des requêtes LINQ, avec les implications que cela comporte sur les performances de requête.

En dehors de LINQ, vous avez appris une technique de tour de cartes utilisée par les magiciens. Les magiciens utilisent le mélange faro pour pouvoir contrôler le déplacement de chaque carte dans le jeu. Maintenant que vous le savez, gardez le secret !

Pour plus d'informations sur LINQ, voir :

- LINQ (Language-Integrated Query)
- Introduction à LINQ
- Opérations de requête LINQ de base (C#)
- Transformations de données avec LINQ (C#)
- Syntaxe de requête et syntaxe de méthode dans LINQ (C#)
- Fonctionnalités C# qui prennent en charge LINQ

Utiliser des attributs en C#

02/11/2020 • 14 minutes to read • Edit Online

Les attributs fournissent un moyen permettant d'associer des informations au code de manière déclarative. Ils peuvent également fournir un élément réutilisable qui peut être appliqué à diverses cibles.

Considérez l'attribut `[Obsolete]`. Vous pouvez l'appliquer aux classes, structures, méthodes, constructeurs et bien plus encore. Il *déclare* que l'élément est obsolète. Il revient ensuite au compilateur C# de rechercher cet attribut et d'effectuer des actions en réponse.

Dans ce didacticiel, vous allez voir comment ajouter des attributs à votre code, comment créer et utiliser vos propres attributs et comment utiliser des attributs qui sont intégrés à .NET Core.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET core. Vous trouverez les instructions d'installation dans la page téléchargements .net Core . Vous pouvez exécuter cette application sur Windows, Ubuntu Linux, Mac OS ou dans un conteneur Docker. Vous devez installer l'éditeur de code de votre choix. Les descriptions ci-dessous utilisent Visual Studio Code, un éditeur open source et multiplateforme. Cependant, vous pouvez utiliser les outils avec lesquels vous êtes le plus à l'aise.

Création de l'application

Maintenant que vous avez installé tous les outils, créez une nouvelle application .NET Core. Pour utiliser le générateur de ligne de commande, exécutez la commande suivante dans votre interpréteur de commandes préféré :

```
dotnet new console
```

Cette commande permet de créer des fichiers de projet .NET Core. Vous devez exécuter `dotnet restore` pour restaurer les dépendances nécessaires à la compilation de ce projet.

Vous n'avez pas à exécuter `dotnet restore`, car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration, comme,,, `dotnet new` `dotnet build` `dotnet run` `dotnet test` `dotnet publish` et `dotnet pack`. Pour désactiver la restauration implicite, utilisez l' `--no-restore` option.

La `dotnet restore` commande est toujours utile dans certains scénarios où la restauration explicite est appropriée, comme les Builds d'intégration continue dans Azure DevOps services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

Pour plus d'informations sur la gestion des flux NuGet, consultez la `dotnet restore` documentation.

Pour exécuter le programme, utilisez `dotnet run`. Vous devriez voir le résultat dans la « Hello, World » dans la console.

Guide pratique d'ajout d'attributs aux propriétés

En C#, les attributs sont des classes qui héritent de la classe de base `Attribute`. Toute classe qui hérite de `Attribute` peut être utilisée comme une sorte de « balise » sur les autres éléments de code. Par exemple, il existe un attribut appelé `ObsoleteAttribute`. Il sert à signaler que le code est obsolète et ne doit pas être plus utilisé. Vous pouvez placer cet attribut sur une classe, par exemple en utilisant des crochets.

```
[Obsolete]
public class MyClass
{
}
```

Notez que si la classe est nommée `ObsoleteAttribute`, il est seulement nécessaire d'utiliser `[Obsolete]` dans le code. Il s'agit d'une convention que C# respecte. Vous pouvez utiliser le nom complet `[ObsoleteAttribute]` si vous le souhaitez.

Lorsque vous marquez une classe comme étant obsolète, il est judicieux de fournir des informations indiquant *pourquoi* elle est obsolète, et/ou *quelle classe* utiliser à la place. Faites cela en passant un paramètre de chaîne à l'attribut obsolète.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

La chaîne est passée comme argument à un constructeur `ObsoleteAttribute`, comme si vous écriviez `var attr = new ObsoleteAttribute("some string")`.

Les paramètres à un constructeur d'attribut sont limités aux types simples/littéraux : `bool, int, double, string, Type, enums, etc` et les tableaux de ces types. Vous ne pouvez pas utiliser une expression ou une variable. Vous êtes libre d'utiliser des paramètres positionnels ou nommés.

Guide pratique de création de votre propre attribut

La création d'un attribut est aussi simple que l'héritage de la classe de base `Attribute`.

```
public class MySpecialAttribute : Attribute
{
}
```

Avec les éléments ci-dessus, je peux maintenant utiliser `[MySpecial]` (ou `[MySpecialAttribute]`) en tant qu'attribut ailleurs dans la base de code.

```
[MySpecial]
public class SomeOtherClass
{
}
```

Les attributs dans la bibliothèque de classes de base .NET, comme `ObsoleteAttribute`, déclenchent certains comportements au sein du compilateur. Toutefois, un attribut que vous créez agit uniquement comme métadonnées et ne produit pas de code dans la classe d'attributs en cours d'exécution. Il revient à vous d'agir sur ces métadonnées ailleurs dans votre code (plus loin dans ce didacticiel).

Il y a ici un « piège » à éviter. Comme mentionné ci-dessus, seuls certains types sont autorisés à être passés comme arguments lors de l'utilisation d'attributs. Toutefois, lorsque vous créez un type d'attribut, le compilateur C# ne vous empêche pas de créer ces paramètres. Dans l'exemple ci-dessous, j'ai créé un attribut avec un constructeur qui se compile correctement.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```

Toutefois, vous ne pourrez pas utiliser ce constructeur avec une syntaxe d'attribut.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

L'exemple ci-dessus entraîne une erreur du compilateur, comme
`Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

Comment limiter l'utilisation d'attributs

Les attributs peuvent être utilisés sur plusieurs « cibles ». Les exemples ci-dessus les illustrent sur des classes, mais ils peuvent également être utilisés sur :

- Assembly
- Classe
- Constructeur
- Délégué
- Énumération
- Événement
- Champ
- GenericParameter
- Interface
- Méthode
- Module
- Paramètre
- Propriété
- ReturnValue
- Structure

Lorsque vous créez une classe d'attributs, par défaut, C# vous permettra d'utiliser cet attribut sur une des cibles d'attribut possibles. Si vous souhaitez limiter votre attribut à certaines cibles, vous pouvez le faire à l'aide de `AttributeUsageAttribute` sur votre classe d'attributs. C'est exact, un attribut sur un attribut !

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Si vous tentez de placer l'attribut ci-dessus sur quelque chose qui n'est pas une classe ou un struct, vous obtiendrez une erreur du compilateur, comme
`Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Guide d'utilisation des attributs associés à un élément de code

Les attributs agissent comme métadonnées. Sans une certaine force extérieure, vous ne les verrez pas faire grand-chose.

Pour rechercher et agir sur des attributs, la Réflexion est généralement nécessaire. Je n'aborderai pas la réflexion de façon approfondie dans ce didacticiel, mais l'idée fondamentale est que la réflexion vous permet d'écrire du code en C# qui examine un autre code.

Par exemple, vous pouvez utiliser la réflexion pour obtenir des informations sur une classe (ajoutez `using System.Reflection;` au début de votre code) :

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

Cela imprimera quelque chose comme :

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

Une fois que vous avez un objet `TypeInfo` (ou `MemberInfo`, `FieldInfo`, etc.), vous pouvez utiliser la méthode `GetCustomAttributes`. Cette opération renvoie une collection d'objets `Attribute`. Vous pouvez également utiliser `GetCustomAttribute` et spécifier un type d'attribut.

Voici un exemple d'utilisation de `GetCustomAttributes` sur une instance de `MemberInfo` pour `MyClass` (avec un attribut `[Obsolete]` associé, comme vu précédemment).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Cela imprimera ceci sur la console : `Attribute on MyClass: ObsoleteAttribute`. Essayez d'ajouter d'autres attributs à `MyClass`.

Il est important de noter que ces objets `Attribute` sont instanciés de manière différée. Autrement dit, ils ne sont pas instanciés jusqu'à ce que vous utilisiez `GetCustomAttribute` ou `GetCustomAttributes`. Ils sont également instanciés chaque fois. Appeler `GetCustomAttributes` deux fois à la suite renverra deux instances différentes de `ObsoleteAttribute`.

Attributs communs dans la bibliothèque de classes de base (BCL)

Les attributs sont utilisés par de nombreux outils et frameworks. NUnit utilise des attributs tels que `[Test]` et `[TestFixture]`, qui sont utilisés par le Test Runner de NUnit. ASP.NET MVC utilise des attributs tels que `[Authorize]` et fournit un framework de filtres d'action pour traiter les problèmes transversaux sur les actions MVC. PostSharp utilise la syntaxe d'attribut pour permettre la programmation orientée aspect en C#.

Voici quelques attributs importants générés dans les bibliothèques de classe de base de .NET Core :

- `[Obsolete]`. Celui-ci a été utilisé dans les exemples ci-dessus, et il se trouve dans l'espace de noms `System`. Il est utile de fournir des informations déclaratives sur une base de code changeante. Un message peut être fourni sous la forme d'une chaîne et un autre paramètre booléen peut servir à passer d'un avertissement du compilateur à une erreur du compilateur.

- `[Conditional]`. Cet attribut se trouve dans l'espace de noms `System.Diagnostics`. Cet attribut peut être appliqué aux méthodes (ou classes d'attributs). Vous devez transmettre une chaîne au constructeur. Si cette chaîne ne correspond pas à une directive `#define`, les appels à cette méthode (mais pas la méthode proprement dite) sont supprimés par le compilateur C#. Cela est généralement utilisé pour le débogage (diagnostic).

- `[CallerMemberName]`. Cet attribut peut être utilisé sur les paramètres et se trouve dans l'espace de noms `System.Runtime.CompilerServices`. Il s'agit d'un attribut utilisé pour injecter le nom de la méthode qui appelle une autre méthode. Cela est généralement utilisé comme moyen d'éliminer les « chaînes magiques » lors de l'implémentation de INotifyPropertyChanged dans divers frameworks d'interface utilisateur. Par exemple :

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

Dans le code ci-dessus, il est inutile d'avoir une chaîne littérale `"Name"`. Cela peut vous aider à éviter les bogues liés aux fautes de frappe et rend également plus simple la refactorisation/les changements de nom.

Résumé

Si les attributs offrent à C# la puissance déclarative, ils constituent une forme de code de type métadonnées et n'agissent pas par eux-mêmes.

Nouveautés dans C# 9.0

16/12/2020 • 34 minutes to read • Edit Online

C# 9,0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Enregistrements
- Setter init uniquement
- Instructions de niveau supérieur
- Améliorations des critères spéciaux
- Entiers dimensionnés natifs
- Pointeurs fonction
- Supprimer l'émission de l'indicateur localsinit
- Nouvelles expressions typées cibles
- fonctions anonymes statiques
- Expressions conditionnelles typées cible
- Types de retour covariant
- `GetEnumerator` Prise en charge des extensions pour les `foreach` boucles
- Paramètres d'abandon lambda
- Attributs sur des fonctions locales
- Initialiseurs de module
- Nouvelles fonctionnalités pour les méthodes partielles

C# 9,0 est pris en charge sur **.net 5**. Pour plus d'informations, consultez contrôle de version du langage C#.

Vous pouvez télécharger le dernier Kit de développement logiciel (SDK) .NET à partir de la page de téléchargements .net.

Types d'enregistrements

C# 9,0 introduit *_types d'enregistrements_*, qui sont un type référence qui fournit des méthodes synthétisées pour fournir une sémantique de valeur pour l'égalité. Les enregistrements sont immuables par défaut.

Les types d'enregistrements facilitent la création de types référence immuables dans .NET. Historiquement, les types .NET sont largement classés comme types référence (y compris les classes et les types anonymes) et les types valeur (y compris les structs et les tuples). Bien que les types de valeurs immuables soient recommandés, les types valeur mutables n'introduisent pas souvent des erreurs. Les variables de type valeur contiennent les valeurs afin que les modifications soient apportées à une copie des données d'origine lorsque les types valeur sont passés aux méthodes.

Il existe également de nombreux avantages par rapport aux types de référence immuables. Ces avantages sont plus prononcés dans les programmes simultanés avec des données partagées. Malheureusement, C# vous obligeait à écrire un peu de code supplémentaire pour créer des types de référence immuables. Les enregistrements fournissent une déclaration de type pour un type référence immuable qui utilise une sémantique de valeur pour l'égalité. Les méthodes synthétisées pour l'égalité et les codes de hachage considèrent deux enregistrements comme égaux si leurs propriétés sont toutes égales. Tenez compte de cette définition :

```
public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) => (FirstName, LastName) = (first, last);
}
```

La définition d'enregistrement crée un `Person` type qui contient deux propriétés ReadOnly : `FirstName` et `LastName` . Le `Person` type est un type référence. Si vous avez regardé le langage intermédiaire, il s'agit d'une classe. Elle est immuable dans la mesure où aucune des propriétés ne peut être modifiée une fois qu'elle a été créée. Lorsque vous définissez un type d'enregistrement, le compilateur synthétise plusieurs autres méthodes pour vous :

- Méthodes pour les comparaisons d'égalité basées sur des valeurs
- Remplacer pour GetHashCode()
- Copier et cloner des membres
- `PrintMembers` et ToString()

Les enregistrements prennent en charge l'héritage. Vous pouvez déclarer un nouvel enregistrement dérivé de `Person` comme suit :

```
public record Teacher : Person
{
    public string Subject { get; }

    public Teacher(string first, string last, string sub)
        : base(first, last) => Subject = sub;
}
```

Vous pouvez également sceller les enregistrements pour éviter toute dérivation supplémentaire :

```
public sealed record Student : Person
{
    public int Level { get; }

    public Student(string first, string last, int level) : base(first, last) => Level = level;
}
```

Le compilateur synthétise différentes versions des méthodes ci-dessus. Les signatures de méthode dépendent de si le type d'enregistrement est sealed et si la classe de base directe est Object. Les enregistrements doivent avoir les fonctionnalités suivantes :

- L'égalité est basée sur la valeur et comprend une vérification de la correspondance des types. Par exemple, un `Student` ne peut pas être égal à un `Person` , même si les deux enregistrements partagent le même nom.
- Une représentation sous forme de chaîne cohérente est générée pour les enregistrements.
- Les enregistrements prennent en charge la construction de copie. La bonne construction de copie doit inclure des hiérarchies d'héritage et des propriétés ajoutées par les développeurs.
- Les enregistrements peuvent être copiés avec modification. Ces opérations de copie et de modification prennent en charge la mutation non destructrice.

En plus des `Equals` surcharges familières, `operator ==` et `operator !=` , le compilateur synthétise une nouvelle `EqualityContract` propriété. La propriété retourne un `Type` objet qui correspond au type de l'enregistrement. Si le type de base est `object` , la propriété est `virtual` . Si le type de base est un autre type d'enregistrement, la propriété est un `override` . Si le type d'enregistrement est `sealed` , la propriété est `sealed` . La synthèse

`GetHashCode` utilise le `GetHashCode` de toutes les propriétés et les champs déclarés dans le type de base et le type d'enregistrement. Ces méthodes synthétisées appliquent l'égalité basée sur les valeurs dans une hiérarchie d'héritage. Cela signifie qu'un `Student` ne sera jamais considéré comme égal à un `Person` portant le même nom. Les types des deux enregistrements doivent correspondre, et toutes les propriétés partagées entre les types d'enregistrements sont égales.

Les enregistrements ont également un constructeur synthétisé et une méthode de clonage pour créer des copies. Le constructeur synthétisé a un seul paramètre du type d'enregistrement. Il génère un nouvel enregistrement avec les mêmes valeurs pour toutes les propriétés de l'enregistrement. Ce constructeur est privé si l'enregistrement est sealed, sinon il est protégé. La méthode « Clone » synthétisée prend en charge la construction de copie pour les hiérarchies d'enregistrement. Le terme « clone » est placé entre guillemets, car le nom réel est généré par le compilateur. Vous ne pouvez pas créer une méthode nommée `Clone` dans un type d'enregistrement. La méthode « Clone » synthétisée renvoie le type d'enregistrement copié à l'aide de la répartition virtuelle. Le compilateur ajoute des modificateurs différents pour la méthode « Clone » en fonction des modificateurs d'accès sur le `record` :

- Si le type d'enregistrement est `abstract` , la méthode « Clone » est également `abstract` . Si le type de base n'est pas `object` , la méthode est également `override` .
- Pour les types d'enregistrements qui ne sont pas `abstract` lorsque le type de base est `object` :
  - Si l'enregistrement est `sealed` , aucun modificateur supplémentaire n'est ajouté à la méthode « Clone » (ce qui signifie qu'il ne l'est pas `virtual`).
  - Si l'enregistrement n'est pas `sealed` , la méthode « Clone » est `virtual` .
- Pour les types d'enregistrements qui ne sont `abstract` pas lorsque le type de base n'est pas `object` :
  - Si l'enregistrement est `sealed` , la méthode « Clone » est également `sealed` .
  - Si l'enregistrement n'est pas `sealed` , la méthode « Clone » est `override` .

Le résultat de toutes ces règles est l'égalité est implémentée de manière cohérente dans n'importe quelle hiérarchie de types d'enregistrements. Deux enregistrements sont égaux si leurs propriétés sont égales et que leurs types sont identiques, comme indiqué dans l'exemple suivant :

```
var person = new Person("Bill", "Wagner");
var student = new Student("Bill", "Wagner", 11);

Console.WriteLine(student == person); // false
```

Le compilateur synthétise deux méthodes qui prennent en charge la sortie imprimée : une ToString() substitution et `PrintMembers` . `PrintMembers` Prend System.Text.StringBuilder comme argument. Elle ajoute une liste séparée par des virgules de noms et de valeurs de propriété pour toutes les propriétés du type d'enregistrement. `PrintMembers` appelle l'implémentation de base pour tous les enregistrements dérivés d'autres enregistrements. La ToString() substitution retourne la chaîne produite par `PrintMembers` , entourée de `{` et de `}` . Par exemple, la ToString() méthode pour `Student` retourne un `string` semblable au code suivant :

```
"Student { LastName = Wagner, FirstName = Bill, Level = 11 }"
```

Les exemples indiqués jusqu'à présent utilisent la syntaxe traditionnelle pour déclarer des propriétés. Il existe une forme plus concise appelée *enregistrements positionnels*. Voici les trois types d'enregistrements définis précédemment en tant qu'enregistrements positionnels :

```
public record Person(string FirstName, string LastName);

public record Teacher(string FirstName, string LastName,
    string Subject)
    : Person(FirstName, LastName);

public sealed record Student(string FirstName,
    string LastName, int Level)
    : Person(FirstName, LastName);
```

Ces déclarations créent les mêmes fonctionnalités que la version antérieure (avec quelques fonctionnalités supplémentaires décrites dans la section suivante). Ces déclarations se terminent par un point-virgule au lieu de crochets, car ces enregistrements n'ajoutent pas de méthodes supplémentaires. Vous pouvez ajouter un corps et inclure également des méthodes supplémentaires :

```
public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}
```

Le compilateur produit une `Deconstruct` méthode pour les enregistrements positionnels. La `Deconstruct` méthode a des paramètres qui correspondent aux noms de toutes les propriétés publiques dans le type d'enregistrement. La `Deconstruct` méthode peut être utilisée pour déconstruire l'enregistrement dans ses propriétés de composant :

```
var person = new Person("Bill", "Wagner");

var (first, last) = person;
Console.WriteLine(first);
Console.WriteLine(last);
```

Enfin, les enregistrements prennent en charge les `with` expressions. Un * `with` expression_ _ *indique au compilateur de créer une copie d'un enregistrement, mais _with* propriétés spécifiées ont été modifiées :

```
Person brother = person with { FirstName = "Paul" };
```

La ligne précédente crée un nouvel `Person` enregistrement où la `LastName` propriété est une copie de `person` , et `FirstName` est `"Paul"` . Vous pouvez définir n'importe quel nombre de propriétés dans une `with` expression. Vous pouvez également utiliser des `with` expressions pour créer une copie exacte. Vous spécifiez le jeu vide pour les propriétés à modifier :

```
Person clone = person with { };
```

Tous les membres synthétisés, à l'exception de la méthode « Clone », peuvent être écrits par vous-même. Si un type d'enregistrement a une méthode qui correspond à la signature d'une méthode synthétisée, le compilateur ne synthétise pas cette méthode. L' `Dog` exemple d'enregistrement précédent contient une méthode codée à la main ToString() à titre d'exemple.

En savoir plus sur les types d'enregistrements dans ce didacticiel sur l' exploration des enregistrements .

Setter init uniquement

*Les **Setters init uniquement** _ fournissent une syntaxe cohérente pour initialiser les membres d'un objet. Les initialiseurs de propriété permettent de savoir clairement quelle valeur définit la propriété. L'inconvénient est que ces propriétés doivent pouvoir être définies. À compter de C# 9,0, vous pouvez créer des `init` accesseurs au lieu d' `set` accesseurs pour les propriétés et les indexeurs. Les appelants peuvent utiliser la syntaxe de l'initialiseur de propriété pour définir ces valeurs dans les expressions de création, mais ces propriétés sont en lecture seule une fois que la construction est terminée. Les Setters init uniquement fournissent une fenêtre pour modifier l'État. Cette fenêtre se ferme à la fin de la phase de construction. La phase de construction se termine effectivement après toute initialisation, y compris les initialiseurs de propriété et les expressions with-.

Vous pouvez déclarer `init` uniquement des accesseurs set dans n'importe quel type que vous écrivez. Par exemple, le struct suivant définit une structure d'observation météorologique :

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

Les appelants peuvent utiliser la syntaxe de l'initialiseur de propriété pour définir les valeurs, tout en préservant l'immuabilité :

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

Toutefois, la modification d'une observation après l'initialisation est une erreur en affectant à une propriété init-only en dehors de l'initialisation :

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Les Setters init uniquement peuvent être utiles pour définir des propriétés de classe de base à partir de classes dérivées. Ils peuvent également définir des propriétés dérivées par le biais d'assistances dans une classe de base. Les enregistrements positionnels déclarent des propriétés à l'aide des accesseurs set init only. Ces accesseurs set sont utilisés dans les expressions with. Vous pouvez déclarer des Setters init uniquement pour n'importe quel `class` ou `struct` vous définissez.

Instructions de niveau supérieur

Les instructions de niveau supérieur suppriment la cérémonie inutile de nombreuses applications. Prenons le « Hello World ! » canonique. table

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Il n'y a qu'une seule ligne de code qui fait quoi que ce soit. Avec les instructions de niveau supérieur, vous pouvez remplacer tout ce qui est réutilisable par l' `using` instruction et la ligne unique qui effectue le travail :

```
using System;

Console.WriteLine("Hello World!");
```

Si vous souhaitez un programme à une seule ligne, vous pouvez supprimer la `using` directive et utiliser le nom de type complet :

```
System.Console.WriteLine("Hello World!");
```

Un seul fichier de votre application peut utiliser des instructions de niveau supérieur. Si le compilateur trouve des instructions de niveau supérieur dans plusieurs fichiers sources, il s'agit d'une erreur. C'est également une erreur si vous combinez des instructions de niveau supérieur avec une méthode de point d'entrée de programme déclarée, généralement une `Main` méthode. Dans un sens, vous pouvez penser qu'un fichier contient les instructions qui seraient normalement dans la `Main` méthode d'une `Program` classe.

L'une des utilisations les plus courantes de cette fonctionnalité est la création de documents pédagogiques. Les développeurs C# débutants peuvent écrire le « Hello World ! » canonique dans une ou deux lignes de code. Aucune des cérémonie supplémentaires n'est nécessaire. Toutefois, les développeurs chevronnés trouveront de nombreuses utilisations pour cette fonctionnalité également. Les instructions de niveau supérieur permettent une expérience de type script pour les expérimentations similaires à celles fournies par les blocs-notes Jupyter. Les instructions de niveau supérieur sont idéales pour les petits programmes et utilitaires de console. Les Azure Functions sont un cas d'usage idéal pour les instructions de niveau supérieur.

Plus important encore, les instructions de niveau supérieur ne limitent pas la portée ou la complexité de votre application. Ces instructions peuvent accéder ou utiliser n'importe quelle classe .NET. Ils ne limitent pas non plus l'utilisation des arguments de ligne de commande ou des valeurs de retour. Les instructions de niveau supérieur peuvent accéder à un tableau de chaînes nommées args. Si les instructions de niveau supérieur retournent une valeur entière, cette valeur devient le code de retour de l'entier d'une méthode synthétisée `Main` . Les instructions de niveau supérieur peuvent contenir des expressions Async. Dans ce cas, le point d'entrée synthétisé retourne un `Task` , ou `Task<int>` .

Améliorations des critères spéciaux

C# 9 comprend de nouvelles améliorations de la mise en correspondance des modèles :

- Les *modèles de type* correspondent à une variable est un type

- Les *modèles entre parenthèses* appliquent ou mettent en évidence la précédence des combinaisons de modèles
- *Les* `and` *modèles conjonctives* nécessitent que les deux modèles correspondent
- *Les* `or` *modèles disjonctive* nécessitent un modèle de correspondance
- Les *`not` modèles de négation_*_ requièrent qu'un modèle ne corresponde pas
- Les *modèles relationnels* requièrent que l'entrée soit inférieure à, supérieure à, inférieure ou égale à une constante donnée, ou supérieure ou égale à celle-ci.

Ces modèles enrichissent la syntaxe des modèles. Prenons les exemples suivants :

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

En guise d'alternative, avec des parenthèses facultatives pour qu'elle soit claire et qu'elle `and` a une priorité plus élevée que `or` :

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

L'une des utilisations les plus courantes est une nouvelle syntaxe pour un contrôle de valeur NULL :

```
if (e is not null)
{
    // ...
}
```

L'un de ces modèles peut être utilisé dans n'importe quel contexte où les modèles sont autorisés : `is` les expressions de modèle, les `switch` expressions, les modèles imbriqués et le modèle de l'étiquette d'une `switch` instruction `case` .

Performances et interopérabilité

Trois nouvelles fonctionnalités améliorent la prise en charge de l'interopérabilité native et des bibliothèques de bas niveau qui requièrent des performances élevées : les entiers de taille native, les pointeurs de fonction et l'omission de l' `localsinit` indicateur.

Les entiers de taille native, `nint` et `nuint` , sont des types entiers. Elles sont exprimées par les types sous-jacents System.IntPtr et System.UIntPtr . Le compilateur recadre les conversions et les opérations supplémentaires pour ces types comme des ints natifs. Les entiers de taille native définissent des propriétés pour `MaxValue` ou `MinValue` . Ces valeurs ne peuvent pas être exprimées comme des constantes de compilation, car elles dépendent de la taille native d'un entier sur l'ordinateur cible. Ces valeurs sont en lecture seule au moment de l'exécution. Vous pouvez utiliser des valeurs constantes pour `nint` dans la plage [`int.MinValue` .. `int.MaxValue`]. Vous pouvez utiliser des valeurs constantes pour `nuint` dans la plage [`uint.MinValue` .. `uint.MaxValue`]. Le compilateur exécute un repli constant pour tous les opérateurs unaires et binaires à l'aide des System.Int32 System.UInt32 types et. Si le résultat ne tient pas dans 32 bits, l'opération est exécutée au moment de l'exécution et n'est pas considérée comme une constante. Les entiers de taille native peuvent augmenter les performances dans les scénarios où les calculs d'entiers sont largement utilisés et doivent avoir les performances les plus rapides possibles.

Les pointeurs fonction fournissent une syntaxe simple pour accéder aux opcodes IL `ldftn` et `calli` . Vous pouvez déclarer des pointeurs de fonction à l'aide de la nouvelle `delegate_` syntaxe. Un `delegate*` type est un type pointeur. L'appel du `delegate*` type utilise `calli` , contrairement à un délégué qui utilise `callvirt` sur la `Invoke()` méthode. Syntaxiquement, les appels sont identiques. L'appel du pointeur de fonction utilise la `managed` Convention d'appel. Vous ajoutez le `unmanaged` mot clé après la `delegate*` syntaxe pour déclarer que vous

souhaitez la `unmanaged` Convention d'appel. D'autres conventions d'appel peuvent être spécifiées à l'aide d'attributs sur la `delegate*` déclaration.

Enfin, vous pouvez ajouter le System.Runtime.CompilerServices.SkipLocalsInitAttribute pour indiquer au compilateur de ne pas émettre l' `localsinit` indicateur. Cet indicateur indique au CLR d'initialiser à zéro toutes les variables locales. L' `localsinit` indicateur a été le comportement par défaut pour C# depuis 1,0. Toutefois, l'initialisation sans zéro supplémentaire peut avoir un impact mesurable sur les performances dans certains scénarios. En particulier, lorsque vous utilisez `stackalloc` . Dans ce cas, vous pouvez ajouter le SkipLocalsInitAttribute . Vous pouvez l'ajouter à une méthode ou une propriété unique, ou à un `class` , `struct` , `interface` ou même un module. Cet attribut n'affecte pas `abstract` les méthodes ; il affecte le code généré pour l'implémentation.

Ces fonctionnalités peuvent améliorer les performances dans certains scénarios. Elles doivent être utilisées uniquement après un test minutieux avant et après l'adoption. Le code impliquant des entiers de taille native doit être testé sur plusieurs plateformes cibles avec différentes tailles d'entier. Les autres fonctionnalités nécessitent du code non sécurisé.

Fonctionnalités d'ajustement et de fin

La plupart des autres fonctionnalités vous aident à écrire du code plus efficacement. En C# 9,0, vous pouvez omettre le type dans une `new` expression lorsque le type de l'objet créé est déjà connu. L'utilisation la plus courante est dans les déclarations de champ :

```
private List<WeatherObservation> _observations = new();
```

Le type cible `new` peut également être utilisé lorsque vous devez créer un nouvel objet à passer en tant qu'argument à une méthode. Envisagez une `ForecastFor()` méthode avec la signature suivante :

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

Vous pouvez l'appeler comme suit :

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Une autre utilisation intéressante de cette fonctionnalité est de l'associer aux propriétés init only pour initialiser un nouvel objet :

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

Vous pouvez retourner une instance créée par le constructeur par défaut à l'aide d'une `return new();` instruction.

Une fonctionnalité similaire améliore la résolution de type cible des expressions conditionnelles. Avec cette modification, les deux expressions n'ont pas besoin d'une conversion implicite de l'une à l'autre, mais elles peuvent toutes deux avoir des conversions implicites en un type cible. Vous ne remarquerez probablement pas cette modification. Ce que vous remarquerez, c'est que certaines expressions conditionnelles qui nécessitaient auparavant des casts ou ne seraient pas compilées.

À compter de C# 9,0, vous pouvez ajouter le `static` modificateur aux expressions lambda ou aux méthodes anonymes. Les expressions lambda statiques sont analogues aux `static` fonctions locales : une méthode lambda statique ou anonyme ne peut pas capturer les variables locales ou l'état de l'instance. Le `static` modificateur empêche la capture accidentelle d'autres variables.

Les types de retour covariants fournissent la flexibilité pour les types de retour des méthodes override . Une méthode override peut retourner un type dérivé du type de retour de la méthode de base substituée. Cela peut être utile pour les enregistrements et pour d'autres types qui prennent en charge les méthodes de fabrique ou de clonage virtuel.

En outre, la `foreach` boucle reconnaît et utilise une méthode d'extension `GetEnumerator` qui, autrement, satisfait le `foreach` modèle. Ce changement signifie `foreach` qu'il est cohérent avec d'autres constructions basées sur des modèles, telles que le modèle asynchrone et la déconstruction basée sur des modèles. Dans la pratique, cette modification signifie que vous pouvez ajouter la `foreach` prise en charge à n'importe quel type. Vous devez limiter son utilisation à lorsque l'énumération d'un objet est logique dans votre conception.

Ensuite, vous pouvez utiliser des éléments ignorés comme paramètres pour les expressions lambda. Cela vous permet d'éviter de nommer l'argument et le compilateur peut ne pas l'utiliser. Vous utilisez `_` pour n'importe quel argument. Pour plus d'informations, consultez la section paramètres d'entrée d'une expression lambda de l'article expressions lambda .

Enfin, vous pouvez maintenant appliquer des attributs à des fonctions locales. Par exemple, vous pouvez appliquer des annotations d'attribut Nullable aux fonctions locales.

Prise en charge des générateurs de code

Deux fonctionnalités finales prennent en charge les générateurs de code C#. Les générateurs de code C# sont un composant que vous pouvez écrire et qui est semblable à un Roslyn Analyzer ou à un correctif de code. La différence est que les générateurs de code analysent le code et écrivent de nouveaux fichiers de code source dans le cadre du processus de compilation. Un générateur de code standard recherche des attributs ou d'autres conventions dans le code.

Un générateur de code lit des attributs ou d'autres éléments de code à l'aide des API d'analyse Roslyn. À partir de ces informations, il ajoute un nouveau code à la compilation. Les générateurs de source peuvent uniquement ajouter du code ; ils ne sont pas autorisés à modifier le code existant dans la compilation.

Les deux fonctionnalités ajoutées pour les générateurs de code sont les extensions de la **syntaxe de méthode partielle** _ et les *initialiseurs de module*. Tout d'abord, les modifications apportées aux méthodes partielles. Avant C# 9,0, les méthodes partielles sont, `private` mais ne peuvent pas spécifier un modificateur d'accès, ont un `void` retour et ne peuvent pas avoir de `out` paramètres. Ces restrictions signifiaient que si aucune implémentation de méthode n'est fournie, le compilateur supprime tous les appels à la méthode partielle. C# 9,0 supprime ces restrictions, mais exige que les déclarations de méthode partielles aient une implémentation. Les générateurs de code peuvent fournir cette implémentation. Pour éviter d'introduire une modification avec rupture, le compilateur considère toute méthode partielle sans modificateur d'accès pour suivre les anciennes règles. Si la méthode partielle comprend le `private` modificateur d'accès, les nouvelles règles gouvernent cette méthode partielle.

La deuxième nouvelle fonctionnalité pour les générateurs de code est _ *initialiseurs de module* *. Les initialiseurs de module sont des méthodes auxquelles l' ModuleInitializerAttribute attribut est attaché. Ces méthodes sont appelées par le runtime avant tout autre accès aux champs ou appel de méthode dans le module entier. Méthode d'initialiseur de module :

- Doit être statique
- Doit être sans paramètre
- Doit retourner void
- Ne doit pas être une méthode générique
- Ne doit pas être contenu dans une classe générique
- Doit être accessible à partir du module conteneur

Ce dernier point à puce signifie effectivement que la méthode et sa classe conteneur doivent être internes ou publiques. La méthode ne peut pas être une fonction locale.

Nouveautés de C# 8.0

02/11/2020 • 29 minutes to read • Edit Online

C# 8,0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Membres ReadOnly
- Méthodes d'interface par défaut
- Améliorations des critères spéciaux:
  - Expressions de commutateur
  - Modèles de propriétés
  - Modèles de tuples
  - Modèles positionnels
- Utilisation des déclarations
- Fonctions locales statiques
- Structs ref jetables
- Types références Nullables
- Flux asynchrones
- Supprimable asynchrone
- Index et plages
- Assignation de fusion Null
- Types construits non managés
- Stackalloc dans les expressions imbriquées
- Amélioration des chaînes textuelles interpolées

C# 8,0 est pris en charge sur **.net Core 3. x** et **.NET standard 2,1**. Pour plus d'informations, consultez contrôle de version du langage C#.

La suite de cet article décrit brièvement ces fonctionnalités. Lorsque des articles détaillés sont disponibles, des liens vers ces tutoriels et vues d'ensemble sont indiqués. Vous pouvez explorer ces fonctionnalités dans votre environnement à l'aide de l'outil global `dotnet try` :

1. Installez l'outil global dotnet-try.
2. Clonez le référentiel dotnet/try-samples.
3. Définissez le répertoire actuel sur le sous-répertoire *csharp8* pour le référentiel *try-samples*.
4. Exécutez `dotnet try`.

Membres ReadOnly

Vous pouvez appliquer le `readonly` modificateur aux membres d'un struct. Elle indique que le membre ne modifie pas l'État. C'est plus précis que d'appliquer le modificateur `readonly` à une déclaration `struct`. Examinons le struct mutable suivant :

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Comme la plupart des structs, la `ToString()` méthode ne modifie pas l'État. Vous pouvez indiquer cela en ajoutant le modificateur `readonly` à la déclaration de `ToString()` :

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

La modification précédente génère un avertissement du compilateur, car `ToString` accède à la `Distance` propriété, qui n'est pas marquée comme `readonly` suit :

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

Le compilateur vous avertit lorsqu'il a besoin de créer une copie défensive. La `Distance` propriété ne change pas d'État. vous pouvez donc résoudre cet avertissement en ajoutant le `readonly` modificateur à la déclaration :

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notez que le `readonly` modificateur est nécessaire sur une propriété en lecture seule. Le compilateur ne suppose pas que les `get` accesseurs ne modifient pas l'état ; vous devez déclarer `readonly` explicitement. Les propriétés implémentées automatiquement sont une exception. le compilateur traite tous les accesseurs get implémentés automatiquement comme. `readonly` il n'est donc pas nécessaire d'ajouter le `readonly` modificateur aux `X` `Y` Propriétés et.

Le compilateur applique la règle qui `readonly` ne modifie pas l'état des membres. La méthode suivante n'est pas compilée, sauf si vous supprimez le `readonly` modificateur :

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

Cette fonctionnalité vous permet de spécifier votre intention de conception, afin que le compilateur puisse l'appliquer et procéder à des optimisations basées sur cette intention.

Pour plus d'informations, consultez la section `readonly` membres d'instance de l'article types de structures .

Méthodes d'interface par défaut

Vous pouvez désormais ajouter des membres aux interfaces et fournir une implémentation pour ces membres. Cette fonctionnalité de langage permet aux auteurs d'API d'ajouter des méthodes à une interface dans les versions ultérieures sans pour autant nuire à la compatibilité des binaires ou des sources avec les implémentations existantes de cette interface. Les implémentations existantes *héritent* de l'implémentation par défaut. Cette

fonctionnalité permet également à C# d'interagir avec les API ciblant Android ou Swift, qui prennent en charge des fonctionnalités similaires. Les méthodes d'interface par défaut permettent également des scénarios semblables à une fonctionnalité de langage « traits ».

Les méthodes d'interface par défaut affectent de nombreux scénarios et éléments de langage. Ce premier tutoriel couvre la mise à jour d'une interface à l'aide d'implémentations par défaut. D'autres tutoriels et mises à jour de référence seront disponibles avant le lancement général.

Ajout de modèles à différents endroits

Les **critères spéciaux** offrent des outils permettant de produire des fonctionnalités dépendantes de la forme sur des types de données liés mais différents. C# 7,0 a introduit la syntaxe pour les modèles de type et les modèles de constante à l'aide de l' `is` expression et de l' `switch` instruction. Ces fonctionnalités représentaient les premières étapes provisoires de prise en charge de paradigmes de programmation distinguant données et fonctionnalités. Face à la transition du secteur vers de nouveaux microservices et autres architectures cloud, d'autres outils sont nécessaires pour le langage.

C# 8.0 développe ce vocabulaire en offrant la possibilité d'utiliser d'autres expressions de modèle à davantage d'endroits dans le code. Étudiez ces fonctionnalités si vos données et vos fonctionnalités sont séparées. Les critères spéciaux peuvent être intéressants si vos algorithmes dépendent d'un fait autre que le type de runtime d'un objet. Ces techniques représentent un autre moyen d'exprimer des conceptions.

En plus des nouveaux modèles en de nouveaux endroits, C# 8.0 ajoute des **modèles récursifs**. Le résultat d'une expression de modèle est une expression. Un modèle récursif constitue simplement une expression de modèle appliquée à la sortie d'une autre expression de modèle.

Expressions switch

Souvent, une `switch` instruction produit une valeur dans chacun de ses `case` blocs. **Les expressions switch** permettent d'utiliser une syntaxe d'expression plus concise, comprenant moins de mots clés `case` et `break` répétitifs et moins d'accolades. Prenons par exemple l'enum suivant, qui liste les couleurs de l'arc-en-ciel :

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

Si votre application a défini un type `RGBColor` construit à partir des composants `R`, `G` et `B`, vous pouvez convertir une valeur `Rainbow` en valeurs RVB avec la méthode suivante, qui contient une expression switch :

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

Plusieurs améliorations ont ici été apportées à la syntaxe :

- La variable se situe avant le mot clé `switch`. Ce nouvel ordre permet de distinguer visuellement l'expression switch de l'instruction switch.
- Les éléments `case` et `:` sont remplacés par `=>`, plus concis et plus intuitif.
- Le cas `default` est remplacé par un discard `_`.
- Les corps sont des expressions et non des instructions.

Comparez cela avec le code équivalent utilisant l'instruction `switch` classique :

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Modèles de propriétés

Le **modèle de propriété** permet de faire correspondre les propriétés de l'objet examiné. Prenons un site d'e-commerce qui doit calculer les taxes sur les ventes en fonction de l'adresse de l'acheteur. Ce calcul n'est pas une responsabilité fondamentale d'une `Address` classe. Il changera au fil du temps, probablement plus souvent que n'évoluera le format de l'adresse. Le montant des taxes sur les ventes varie selon la propriété `State` de l'adresse. La méthode suivante utilise le modèle de propriété pour calculer les taxes sur les ventes à partir de l'adresse et du prix :

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

Les critères spéciaux offrent une syntaxe concise pour exprimer cet algorithme.

Modèles de tuples

Certains algorithmes dépendent de plusieurs entrées. Les **modèles de tuples** permettent de basculer des unes aux autres en fonction de plusieurs valeurs exprimées sous forme de tuple. Le code suivant montre une expression switch pour le jeu *Pierre-papier-ciseaux* :

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

Les messages indiquent le vainqueur. Le cas discard représente les trois combinaisons pour les égalités, ou d'autres entrées de texte.

Modèles positionnels

Certains types comportent une méthode `Deconstruct` qui décompose ses propriétés en variables discrètes. Lorsqu'une méthode `Deconstruct` est accessible, il est possible d'utiliser des **modèles positionnels** pour inspecter les propriétés de l'objet et de se servir de ces dernières pour un modèle. Considérez la classe `Point` suivante, dont la méthode `Deconstruct` sert à créer des variables discrètes pour `X` et `Y` :

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

De plus, envisagez l'énumération suivante qui représente diverses positions d'un quadrant :

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

La méthode suivante utilise le **modèle positionnel** pour extraire les valeurs de `x` et de `y`. Ensuite, elle utilise une clause `when` pour déterminer le `Quadrant` du point :

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

Le modèle discard dans le switch précédent indique une correspondance lorsque soit `x`, soit `y` est égal à 0, mais pas les deux. Une expression switch doit produire une valeur ou, si aucun des cas ne correspond, lever une exception. Le compilateur génère un avertissement pour vous si vous ne traitez pas tous les cas possibles dans votre expression de commutateur.

Vous pouvez explorer des techniques de critères spéciaux dans ce tutoriel avancé sur les critères spéciaux.

Déclarations using

Une **déclaration using** est une déclaration de variable précédée par le mot clé `using`. Elle indique au compilateur que la variable déclarée doit être supprimée à la fin de la portée englobante. Prenons par exemple le code suivant, qui écrit un fichier texte :

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

Dans l'exemple précédent, le fichier est supprimé une fois l'accolade fermante de la méthode atteinte. C'est la fin de

la portée dans laquelle `file` est déclaré. Le code précédent équivaut au suivant, qui utilise l'instruction using classique :

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

Dans l'exemple précédent, le fichier est supprimé une fois l'accolade fermante associée à l'instruction `using` atteinte.

Dans les deux cas, le compilateur génère l'appel à `Dispose()`. Le compilateur génère une erreur si l'expression dans l' `using` instruction n'est pas supprimable.

Fonctions locales statiques

Vous pouvez maintenant ajouter le `static` modificateur aux fonctions locales pour vous assurer que la fonction locale ne capture pas (référence) les variables de la portée englobante. Cela génère `CS8421`, « A static local function can't contain a reference to <variable> ».

Prenons le code suivant. La fonction locale `LocalFunction` accède à la variable `y`, déclarée dans la portée englobante (la méthode `M`). Par conséquent, `LocalFunction` ne peut pas être déclarée avec le modificateur `static` :

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

Le code suivant contient une fonction locale statique. Elle peut être statique, car elle n'accède à aucune variable dans la portée englobante :

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Structs ref jetables

Un `struct` déclaré avec le `ref` modificateur ne peut pas implémenter d'interfaces et ne peut donc pas implémenter IDisposable . Par conséquent, pour qu'un `ref struct` soit supprimable, il doit avoir une méthode `void Dispose()` accessible. Cette fonctionnalité s'applique également aux `readonly ref struct` déclarations.

Types références Nullables

Dans un contexte d'annotation Nullable, toute variable d'un type référence est considérée comme un **type référence n'acceptant pas la valeur Null**. Si vous souhaitez indiquer qu'une variable peut être Null, ajoutez `?` au nom de type pour la déclarer comme **type référence Nullable**.

Avec les types références n'acceptant pas la valeur Null, le compilateur utilise l'analyse de flux pour que les variables locales soient initialisées à une valeur non Null une fois déclarées. Les champs doivent être initialisés à la construction. Le compilateur génère un avertissement si la variable n'est pas définie par un appel à l'un des constructeurs disponibles ou par un initialiseur. Par ailleurs, les types références n'acceptant pas la valeur Null ne peuvent pas avoir de valeur pouvant être Null.

Les types références Nullables font l'objet d'aucun contrôle visant à vérifier qu'ils ne sont pas affectés ou initialisées sur Null. Toutefois, le compilateur utilise l'analyse de flux pour comparer à Null toutes les variables d'un type référence Nullable avant d'y accéder ou de lui affecter un type référence n'acceptant pas la valeur Null.

Plus d'informations sur la fonctionnalité, voir la vue d'ensemble des types références Nullables. Essayez par vous-même dans une nouvelle application avec ce tutoriel des types références Nullables. Pour plus d'informations sur le processus de migration d'un codebase dans l'objectif d'utiliser des types références Nullables, voir le tutoriel Migrer une application pour utiliser des types références Nullables.

Flux asynchrones

À partir de C# 8.0, il est possible de créer et de consommer des flux de façon asynchrone. Une méthode qui retourne un flux asynchrone comporte trois propriétés :

1. Elle est déclarée avec le modificateur `async`.
2. Elle retourne un IAsyncEnumerable<T>.
3. La méthode contient des instructions `yield return` pour retourner des éléments consécutifs dans le flux asynchrone.

Pour pouvoir consommer un flux asynchrone, il faut ajouter le mot clé `await` avant le mot clé `foreach` au moment d'énumérer les éléments du flux. Le mot clé `await` exige que la méthode qui énumère le flux asynchrone soit déclarée avec le modificateur `async` et retourne un type autorisé pour une méthode `async`, soit en général Task ou Task<TResult>. Il peut également s'agir de ValueTask ou ValueTask<TResult>. Une méthode peut consommer et produire un flux asynchrone ; elle retournerait alors un IAsyncEnumerable<T>. Le code suivant génère une séquence de 0 à 19, en attendant 100 ms entre chaque nombre :

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Pour énumérer la séquence, on utilise l'instruction `await foreach` :

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

Vous pouvez essayer par vous-même les flux asynchrones dans notre tutoriel Créer et consommer des flux asynchrones. Par défaut, les éléments de flux sont traités dans le contexte capturé. Si vous souhaitez désactiver la capture du contexte, utilisez la TaskAsyncEnumerableExtensions.ConfigureAwait méthode d'extension. Pour plus d'informations sur les contextes de synchronisation et la capture du contexte actuel, consultez l'article sur l' utilisation du modèle asynchrone basésur des tâches.

Supprimable asynchrone

À compter de C# 8,0, le langage prend en charge les types jetables asynchrones qui implémentent l' System.IAsyncDisposable interface. Vous utilisez l' `await using` instruction pour travailler avec un objet supprimé de manière asynchrone. Pour plus d'informations, consultez l'article mettre en œuvre une méthode DisposeAsync .

Index et plages

Les index et les plages fournissent une syntaxe concise pour accéder à des éléments ou des plages uniques dans une séquence.

Cette prise en charge de langage s'appuie sur deux nouveaux types et deux nouveaux opérateurs :

- System.Index représente un index au sein d'une séquence.
- Index de l'opérateur End `^` , qui spécifie qu'un index est relatif à la fin de la séquence.
- System.Range représente une sous-plage d'une séquence.
- Opérateur de plage `..` , qui spécifie le début et la fin d'une plage comme opérandes.

Commençons par les règles concernant les index. Prenons pour exemple un tableau `sequence`. L'index `0` est identique à l'index `sequence[0]`. L'index `^0` est identique à l'index `sequence[sequence.Length]`. Notez que `sequence[^0]` lève une exception, tout comme `sequence[sequence.Length]`. Pour n'importe quel nombre `n`, l'index `^n` est identique à l'index `sequence.Length - n`.

Une plage spécifie son *début* et sa *fin*. Le début de la plage est compris, mais la fin de la plage est exclusive, ce qui signifie que le *début* est inclus dans la plage, mais que la *fin* n'est pas incluse dans la plage. La plage `[0..^0]` représente la plage dans son intégralité, tout comme `[0..sequence.Length]` représente la plage entière.

Prenons quelques exemples. Examinez le tableau suivant, annoté avec son index à partir du début et de la fin :

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Vous pouvez récupérer le dernier mot avec l'index `^1` :

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

Le code suivant crée une sous-plage qui comporte les mots « quick », « brown » et « fox » et va de `words[1]` à `words[3]`. L'élément `words[4]` n'est pas dans la plage.

```
var quickBrownFox = words[1..4];
```

Le code suivant crée une sous-plage qui comporte « lazy » et « dog » et comprend `words[^2]` et `words[^1]`. L'index de fin `words[^0]` n'est pas inclus :

```
var lazyDog = words[^2..^0];
```

Les exemples suivants créent des plages ouvertes au début, à la fin ou les deux :

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

Vous pouvez également déclarer des plages comme variables :

```
Range phrase = 1..4;
```

La plage peut ensuite être utilisée à l'intérieur des caractères `[` et `]` :

```
var text = words[phrase];
```

Non seulement les tableaux prennent en charge les index et les plages. Vous pouvez également utiliser des index et des plages avec String, Span<T> ou ReadOnlySpan<T> . Pour plus d'informations, consultez prise en charge des types d'index et de plages.

Pour explorer davantage les index et les plages, consultez le tutoriel sur les index et les plages.

Assignation de fusion Null

C# 8,0 introduit l'opérateur d'assignation de fusion Null `??=` . Vous pouvez utiliser l' `??=` opérateur pour assigner la valeur de son opérande droit à son opérande gauche uniquement si l'opérande de gauche prend la valeur `null` .

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

Pour plus d'informations, consultez les = l'article Operators .

Types construits non managés

En C# 7,3 et versions antérieures, un type construit (un type qui comprend au moins un argument de type) ne peut pas être un type non managé. À compter de C# 8,0, un type valeur construit est non managé s'il contient uniquement des champs de types non managés.

Par exemple, étant donné la définition suivante du `Coords<T>` type générique :

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

le `Coords<int>` type est un type non managé dans C# 8,0 et versions ultérieures. Comme pour tout type non managé, vous pouvez créer un pointeur vers une variable de ce type ou allouer un bloc de mémoire sur la pile pour les instances de ce type :

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

Pour plus d'informations, consultez types non managés.

Stackalloc dans les expressions imbriquées

À compter de C# 8,0, si le résultat d'une expression stackalloc est du System.Span<T> System.ReadOnlySpan<T> type ou, vous pouvez utiliser l' `stackalloc` expression dans d'autres expressions :

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

Amélioration des chaînes textuelles interpolées

L'ordre des `$` `@` jetons et dans les chaînes textuelles interpolées peut être any : `$@"..."` et `@$"..."` sont des chaînes textuelles interpolées valides. Dans les versions antérieures de C#, le `$` jeton doit apparaître avant le `@` jeton.

Nouveautés de C# 7,0 par C# 7,3

02/11/2020 • 47 minutes to read • Edit Online

C# 7,0 à C# 7,3 a apporté un certain nombre de fonctionnalités et des améliorations incrémentielles à votre expérience de développement avec C#. Cet article fournit une vue d'ensemble des nouvelles fonctionnalités de langage et des options du compilateur. Les descriptions décrivent le comportement de C# 7,3, qui est la version la plus récente prise en charge pour les applications .NET Framework.

L'élément de configuration de sélection de la version de langage a été ajouté avec C# 7,1, ce qui vous permet de spécifier la version du langage du compilateur dans votre fichier projet.

C# 7.0-7.3 ajoute ces fonctionnalités et thèmes au langage C# :

- Tuples et éléments ignorés
  - Vous pouvez créer des types légers et sans nom qui contiennent plusieurs champs publics. Les compilateurs et les outils de l'IDE comprennent la sémantique de ces types.
  - Les éléments ignorés sont les variables temporaires en écriture seule utilisées dans les attributions quand vous ne vous souciez pas de la valeur assignée. Ils s'avèrent utiles lors de la déconstruction de tuples et de types définis par l'utilisateur, ainsi que lors de l'appel de méthodes avec des paramètres `out`.
- Critères spéciaux
  - Vous pouvez créer une logique de branchement basée sur des types arbitraires et les valeurs des membres de ces types.
- `async``Main` méthode
  - Le point d'entrée pour une application peut avoir le modificateur `async`.
- Fonctions locales
  - Vous pouvez imbriquer des fonctions dans d'autres fonctions afin de limiter leur portée et leur visibilité.
- Autres membres expression-bodied
  - La liste des membres pouvant être créés à l'aide d'expressions s'est allongée.
- `throw` Manifestations
  - Vous pouvez lever des exceptions dans les constructions de code qui n'étaient pas autorisées auparavant, car `throw` était une instruction.
- `default` expressions littérales
  - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Améliorations de la syntaxe littérale numérique
  - De nouveaux jetons améliorent la lisibilité des constantes numériques.
- `out` variables
  - Vous pouvez déclarer des valeurs `out` inline comme arguments de la méthode dans laquelle elles sont utilisées.
- Arguments nommés non placés en position de fin
  - Les arguments nommés peuvent être suivis par des arguments de position.
- `private protected` modificateur d'accès
  - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Résolution de surcharge améliorée
  - Nouvelles règles pour résoudre l'ambiguïté de résolution de surcharge.

- Techniques d'écriture de code safe et efficace
  - Une combinaison des améliorations de la syntaxe qui permettent d'utiliser les types valeur avec la sémantique de référence.

Enfin, le compilateur a de nouvelles options :

- `-refout` et `-refonly` qui contrôlent la génération de l' assembly de référence.
- `-publicsign` pour activer la signature d'assemblys Open Source Software (OSS).
- `-pathmap` pour fournir un mappage des répertoires sources.

Le reste de cet article présente une vue d'ensemble de chaque fonctionnalité. Pour chaque fonctionnalité, vous apprendrez le raisonnement sous-jacent et la syntaxe. Vous pouvez explorer ces fonctionnalités dans votre environnement à l'aide de l'outil global `dotnet try` :

1. Installez l'outil global dotnet-try.
2. Clonez le référentiel dotnet/try-samples.
3. Définissez le répertoire actuel sur le sous-répertoire *csharp7* pour le référentiel *try-samples*.
4. Exécuter `dotnet try`.

Tuples et éléments ignorés

C# fournit une syntaxe complète pour les classes et les structs, utilisée pour expliquer l'intention de votre conception. Cependant, cette syntaxe complète nécessite parfois un travail supplémentaire avec peu d'avantages. Vous pouvez souvent écrire des méthodes qui nécessitent une structure simple contenant plusieurs éléments de données. Pour prendre en charge ces scénarios, des *tuples* ont été ajoutées à C#. Les tuples sont des structures de données légères contenant plusieurs champs pour représenter les membres de données. Les champs ne sont pas validés et vous ne pouvez pas définir vos propres méthodes. Les types de tuple C# prennent en charge `==` et `!=` . Pour plus d'informations, consultez

> **NOTE**
> Les tuples étaient disponibles avant C# 7.0, mais ils n'étaient pas efficaces et n'avaient aucune prise en charge du langage. Cela signifiait que les éléments tuples pouvaient uniquement être référencés comme `Item1` , `Item2` , et ainsi de suite. C# 7.0 introduit la prise en charge du langage pour les tuples, ce qui permet d'utiliser des noms sémantiques pour les champs d'un tuple avec de nouveaux types tuple plus efficaces.

Vous pouvez créer un tuple en assignant une valeur à chaque membre et éventuellement, en fournissant des noms sémantiques à chacun des membres du tuple :

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

Le tuple `namedLetters` contient des champs appelés `Alpha` et `Beta` . Ces noms existent uniquement au moment de la compilation et ne sont pas conservés, par exemple lors de l'inspection du tuple à l'aide de la réflexion au moment de l'exécution.

Dans une assignation de tuple, vous pouvez également spécifier les noms des champs dans la partie droite :

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

Dans certains cas, vous pouvez souhaiter décompresser les membres d'un tuple qui ont été retournés à partir d'une méthode. Pour ce faire, déclarez des variables distinctes pour chacune des valeurs dans le tuple. Cette

décompression est appelée *déconstruction* du tuple :

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

Vous pouvez également fournir une déconstruction similaire pour tout type dans le .NET. Vous écrivez une méthode `Deconstruct` en tant que membre de la classe. Cette méthode `Deconstruct` fournit un ensemble d'arguments `out` pour chacune des propriétés que vous voulez extraire. Prenons la classe `Point` suivante qui fournit une méthode de déconstructeur qui extrait les coordonnées `X` et `Y` :

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

Vous pouvez extraire les champs individuels en affectant un `Point` à un tuple :

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

Bien souvent, lorsque vous initialisez un tuple, les variables utilisées pour le côté droit de l'assignation sont les mêmes que les noms que vous souhaitez pour les éléments de tuple : les noms des éléments de tuple peuvent être déduits à partir des variables utilisées pour initialiser le tuple :

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

Pour plus d'informations sur cette fonctionnalité, consultez l'article types de tuples .

Souvent, lors de la déconstruction d'un tuple ou de l'appel d'une méthode avec des paramètres `out`, vous devez définir une variable dont la valeur ne vous importe pas et que vous ne prévoyez pas d'utiliser. C# ajoute la prise en charge des *éléments ignorés* pour gérer ce scénario. Un élément ignoré est une variable en écriture seule dont le nom est `_` (caractère de soulignement) ; vous pouvez assigner toutes les valeurs que vous souhaitez ignorer à la variable unique. Un élément ignoré est semblable à une variable non assignée ; en dehors de l'instruction d'assignation, l'élément ignoré ne peut pas être utilisé dans le code.

Les éléments ignorés sont pris en charge dans les scénarios suivants :

- Lors de la déconstruction de tuples ou de types définis par l'utilisateur.
- Lors d'appels à des méthodes avec des paramètres out.
- Dans une opération de critères spéciaux avec les instructions is et switch.
- Comme un identificateur autonome quand vous voulez explicitement identifier la valeur d'une assignation comme un élément ignoré.

L'exemple suivant définit une méthode `QueryCityDataForYears` qui retourne un tuple à 6 composants qui contient des données pour une ville au cours de deux années différentes. L'appel de méthode dans l'exemple s'intéresse

uniquement à deux valeurs de population retournées par la méthode et traite par conséquent les valeurs restantes dans le tuple comme des éléments ignorés lors de la déconstruction du tuple.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Pour plus d'informations, consultez Éléments ignorés.

Critères spéciaux

Les *critères spéciaux* sont un ensemble de fonctionnalités qui permettent de nouvelles façons d'exprimer le contrôle du workflow dans votre code. Vous pouvez tester des variables pour leur type, leurs valeurs ou les valeurs de leurs propriétés. Ces techniques créent un flot de code plus lisible.

Les critères spéciaux prennent en charge les expressions `is` et les expressions `switch`. L'une ou l'autre permettent d'inspecter un objet et ses propriétés afin de déterminer s'il correspond au modèle recherché. Vous utilisez le mot clé `when` pour spécifier des règles supplémentaires pour le modèle.

L' `is` expression de modèle étend l' `is` opérateur familier pour interroger un objet sur son type et assigner le résultat dans une instruction. Le code suivant vérifie si une variable est un `int`, et si tel est le cas, il l'ajoute à la somme actuelle :

```
if (input is int count)
    sum += count;
```

Le petit exemple précédent illustre les améliorations apportées à l'expression `is`. Vous pouvez tester par rapport à

des types valeur, ainsi que des types référence, et vous pouvez assigner le résultat de réussite à une nouvelle variable du type correct.

L'expression de correspondance switch a une syntaxe familière, basée sur l'instruction `switch` qui fait déjà partie du langage C#. L'instruction switch mise à jour a plusieurs nouvelles constructions :

- Le type directeur d'une expression `switch` n'est plus limité aux types intégraux, aux types `Enum`, `string` ou à un type nullable correspondant à l'un de ces types. Vous pouvez utiliser n'importe quel type.
- Vous pouvez tester le type de l'expression `switch` dans chaque étiquette `case`. Comme avec l'expression `is`, vous pouvez assigner une nouvelle variable à ce type.
- Vous pouvez ajouter une clause `when` pour tester encore les conditions sur cette variable.
- L'ordre des étiquettes `case` est à présent important. La première branche à faire correspondre est exécutée ; les autres sont ignorées.

Le code suivant illustre ces nouvelles fonctionnalités :

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` est le modèle de constante classique.
- `case IEnumerable<int> childSequence:` est un modèle de type.
- `case int n when n > 0:` est un modèle de type avec une autre condition `when`.
- `case null:` est le modèle null.
- `default:` est le cas par défaut classique.

À compter de C# 7.1, l'expression de modèle de `is` et du modèle de type `switch` peut avoir le type d'un paramètre de type générique, ce qui peut se révéler particulièrement utile pour vérifier des types potentiellement `struct` ou `class` tout en évitant le boxing.

Pour plus d'informations sur les critères spéciaux, consultez Critères spéciaux en C#.

Async main

Une méthode *async main* vous permet d'utiliser `await` dans votre méthode `Main`. Auparavant, vous deviez écrire :

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

Vous pouvez désormais écrire :

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

Si votre programme ne retourne pas de code de sortie, vous pouvez déclarer une méthode `Main` qui retourne une Task :

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

Pour plus d'informations, voir l'article async main du guide de programmation.

Fonctions locales

De nombreuses conceptions pour les classes incluent des méthodes qui sont appelées à partir d'un seul emplacement. Ces méthodes privées supplémentaires maintiennent chaque méthode réduite et focalisée. Les *fonctions locales* vous permettent de déclarer des méthodes dans le contexte d'une autre méthode. Il est ainsi plus facile pour les lecteurs de la classe de voir que la méthode locale est appelée uniquement à partir du contexte dans lequel elle a été déclarée.

Il existe deux cas d'utilisation courants pour les fonctions locales : les méthodes iterator publiques et les méthodes async publiques. Ces deux types de méthodes génèrent du code qui signale les erreurs plus tard que ce qu'attendent les programmeurs. Dans les méthodes iterator, toute exception est observée uniquement lors de l'appel de code qui énumère la séquence retournée. Dans les méthodes async, toute exception est observée uniquement quand le `Task` retourné est attendu. L'exemple suivant illustre la séparation entre la validation de paramètres et l'implémentation de l'itérateur à l'aide d'une fonction locale :

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

Il est possible d'utiliser la même technique avec les méthodes `async` pour garantir que les exceptions résultant de la validation d'argument sont levées avant le début de la tâche asynchrone :

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Cette syntaxe est maintenant prise en charge :

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

L'attribut `SomeThingAboutFieldAttribute` est appliqué au champ de stockage généré par le compilateur pour `SomeProperty`. Pour plus d'informations, consultez les attributs dans le guide de programmation C#.

> **NOTE**
>
> Certaines des conceptions prises en charge par les fonctions locales peuvent également être accomplies à l'aide d' *expressions lambda*. Pour plus d'informations, consultez fonctions locales et expressions lambda.

Autres membres expression-bodied

C# 6 a introduit les membres expression-bodied pour les fonctions membres, ainsi que des propriétés en lecture seule. C# 7.0 développe les membres autorisés pouvant être implémentés comme expressions. Dans C# 7.0, vous pouvez implémenter des *constructeurs*, des *finaliseurs* ainsi que des accesseurs `get` et `set` sur des *propriétés* et des *indexeurs*. Le code suivant présente des exemples de chaque élément :

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

> **NOTE**
> Cet exemple n'a pas besoin de finaliseur, mais il est présenté pour illustrer la syntaxe. Vous ne devez implémenter un finaliseur dans votre classe que si cela est nécessaire pour libérer des ressources non managées. Vous devez également envisager d'utiliser la classe SafeHandle au lieu de gérer directement les ressources non managées.

Ces nouveaux emplacements pour les membres expression-bodied représentent une étape importante pour le langage C# : ces fonctionnalités ont été implémentées par des membres de la communauté travaillant sur le projet open source Roslyn.

La modification d'une méthode en un membre expression-bodied est une modification compatible binaire.

Expressions throw

En C#, `throw` a toujours été une instruction. Étant donné que `throw` est une instruction, et non pas une expression, certaines constructions C# ne pouvaient pas l'utiliser. Il s'agit notamment des expressions conditionnelles, des expressions de fusion null, ainsi que de certaines expressions lambda. L'ajout de membres expression-bodied ajoute des emplacements supplémentaires où les expressions `throw` seraient utiles. Afin que vous puissiez écrire l'une de ces constructions, C# 7,0 introduit des *expressions Throw*.

Cet ajout facilite l'écriture de code davantage basé sur des expressions. Vous n'avez pas besoin d'instructions supplémentaires pour la vérification des erreurs.

Expressions littérales par défaut

Les expressions littérales par défaut sont une amélioration des expressions de valeur par défaut. Ces expressions initialisent une variable à la valeur par défaut. Vous deviez écrire auparavant :

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

Vous pouvez désormais omettre le type du côté droit de l'initialisation :

```
Func<string, bool> whereClause = default;
```

Pour plus d'informations, voir la section littéral par défaut de l'article opérateur par défaut.

Améliorations de la syntaxe littérale numérique

La mauvaise interprétation de constantes numériques peut rendre plus difficile la compréhension du code quand il est lu pour la première fois. Les masques de bits ou d'autres valeurs symboliques sont source de méprise. C# 7.0 comprend deux nouvelles fonctionnalités permettant d'écrire des nombres de la manière la plus lisible pour l'utilisation prévue : les *littéraux binaires* et les *séparateurs de chiffres*.

Dans les cas où vous créez des masques de bits chaque fois qu'une représentation binaire d'un nombre rend le code plus lisible, écrivez ce nombre au format binaire :

```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

Le `0b` au début de la constante indique que le nombre est écrit sous la forme d'un nombre binaire. Les nombres binaires peuvent être longs. il est donc souvent plus facile de voir les modèles binaires en introduisant le `_` sous

forme de séparateur numérique, comme indiqué dans la constante binaire de l'exemple précédent. Le séparateur de chiffres peut apparaître n'importe où dans la constante. Pour les nombres de base 10, il est courant de l'utiliser comme séparateur des milliers. Les littéraux numériques hexadécimaux et binaires peuvent commencer par un `_` :

```
public const long BillionsAndBillions = 100_000_000_000;
```

Il est possible d'utiliser le séparateur de chiffres également avec les types `decimal`, `float` et `double` :

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

Globalement, vous pouvez déclarer des constantes numériques avec beaucoup plus de lisibilité.

Variables `out`

La syntaxe existante qui prend en charge les `out` paramètres a été améliorée en C# 7. Vous pouvez désormais déclarer des variables `out` dans la liste d'arguments d'un appel de méthode, au lieu d'écrire une instruction de déclaration distincte :

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

Vous pouvez spécifier le type de la `out` variable pour plus de clarté, comme indiqué dans l'exemple précédent. Toutefois, le langage prend en charge l'utilisation d'une variable locale implicitement typée :

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- Le code est plus facile à lire.
  - Vous déclarez la variable out là où vous l'utilisez, et non sur une ligne de code précédente.
- Il n'est pas nécessaire d'assigner une valeur initiale.
  - En déclarant la variable `out` à l'endroit où elle est utilisée dans un appel de méthode, vous ne pouvez pas l'utiliser accidentellement avant qu'elle soit assignée.

La syntaxe ajoutée dans C# 7.0 pour autoriser les déclarations variables `out` a été étendue pour inclure des initialiseurs de champ, des initialiseurs de propriété, des initialiseurs de constructeur et des clauses de requête. Il permet d'écrire un code tel que l'exemple suivant :

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

Arguments nommés non placés en position de fin

Les appels de méthode peuvent désormais utiliser des arguments nommés qui précèdent les arguments de position quand ces arguments nommés se trouvent dans les positions appropriées. Pour plus d'informations, consultez arguments nommés et facultatifs.

private protected (modificateur d'accès)

Enfin, un nouveau modificateur d'accès composé, `private protected`, indique qu'un membre est accessible à la classe globale ou aux classes dérivées déclarées dans le même assembly. Alors que `protected internal` autorise l'accès par des classes dérivées ou qui se trouvent dans le même assembly, `private protected` limite l'accès aux types dérivés déclarés dans le même assembly.

Pour plus d'informations, consultez modificateurs d'accès dans la référence du langage.

Candidats de surcharge améliorés

Dans chaque version, les règles de résolution de surcharge ont été mises à jour pour résoudre les situations où des appels de méthode ambigus sont face à un choix « évident ». Cette version ajoute trois nouvelles règles pour aider le compilateur à opter pour le choix évident :

1. Lorsqu'un groupe de méthodes contient à la fois une instance et des membres statiques, le compilateur ignore les membres de l'instance si la méthode a été appelée sans récepteur d'instance ou contexte. Le compilateur ignore les membres statiques si la méthode a été appelée avec un récepteur d'instance. Lorsqu'il n'existe aucun récepteur, le compilateur inclut uniquement les membres statiques dans un contexte statique, sinon, à la fois les membres statiques et les membres de l'instance. Lorsque le récepteur représente de façon ambiguë une instance ou un type, le compilateur inclut les deux éléments. Un contexte statique, où un récepteur d'instance `this` implicite ne peut pas être utilisé, inclut le corps des membres où aucun `this` n'est défini, par exemple des membres statiques, ainsi que chaque endroit où `this` ne peut pas être utilisé, par exemple les initialiseurs de champ et les initialiseurs de constructeur.
2. Lorsqu'un groupe de méthodes contient certaines méthodes génériques dont les arguments de type ne répondent pas à leurs contraintes, ces membres sont supprimés de l'ensemble de candidats.
3. Pour une conversion de groupe de méthodes, les méthodes candidates dont le type de retour ne correspond pas à celui du délégué sont supprimées de l'ensemble.

Vous ne remarquerez que cette modification car vous trouverez moins d'erreurs de compilateur pour les surcharges de méthode ambiguës si vous savez quelle méthode est la meilleure.

Code sécurisé plus performant

Vous devez être en mesure d'écrire en toute sécurité un code C# ainsi performant qu'un code non sécurisé. Le code sécurisé évite les classes d'erreurs, par exemple les dépassements de mémoire tampon, les pointeurs perdus et d'autres erreurs d'accès à la mémoire. Ces nouvelles fonctionnalités étendent les fonctionnalités de code sécurisé vérifiables. Efforcez-vous d'utiliser plus de constructions sécurisées pour écrire votre code. Ces fonctionnalités facilitent cette écriture.

Les nouvelles fonctionnalités suivantes prennent en charge le thème de meilleures performances pour le code sécurisé :

- Vous pouvez accéder à des champs fixes sans épinglage.
- Vous pouvez réaffecter des variables locales `ref`.
- Vous pouvez utiliser des initialiseurs sur les tableaux `stackalloc`.
- Vous pouvez utiliser des instructions `fixed` avec tout type prenant en charge un modèle.
- Vous pouvez utiliser des contraintes génériques supplémentaires.
- Le modificateur `in` sur les paramètres pour spécifier qu'un argument est passé par référence, mais pas modifié par la méthode appelée. L'ajout du modificateur `in` à un argument est une modification compatible avec la source.
- Le modificateur `ref readonly` sur les retours de méthode pour indiquer qu'une méthode retourne sa valeur par référence, mais n'autorise pas les écritures sur cet objet. L'ajout du modificateur `ref readonly` est une modification compatible avec la source, si une valeur est assignée au retour. Le fait d'ajouter le modificateur `readonly` à une instruction de retour `ref` existante représente une modification incompatible. Cela nécessite que les appelants mettent à jour de la déclaration de variables locales `ref` pour inclure le modificateur `readonly`.
- La déclaration `readonly struct` pour indiquer qu'un struct est immuable et doit être passé comme un paramètre `in` aux méthodes de ses membres. L'ajout du modificateur `readonly` à une déclaration struct existante est une modification compatible binaire.
- La déclaration `ref struct` pour indiquer qu'un type struct accède directement à la mémoire managée et doit toujours être alloué par la pile. L'ajout du modificateur `ref` à une déclaration `struct` existante est une modification incompatible. Un `ref struct` ne peut pas être membre d'une classe ou utilisé dans d'autres emplacements où il pourrait être alloué sur le tas.

Pour plus d'informations sur toutes ces modifications, voir Écrire du code safe et efficace.

Variables locales et retours ref

Cette fonctionnalité active les algorithmes qui utilisent et retournent des références à des variables définies ailleurs. C'est le cas, par exemple, lors de la recherche d'un emplacement unique comportant certaines caractéristiques dans une matrice de grande taille. La méthode suivante retourne une **référence** à ce stockage dans la matrice :

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Vous pouvez déclarer la valeur de retour en tant que `ref` et modifier cette valeur dans la matrice, comme indiqué dans le code suivant :

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

Le langage C# a plusieurs règles qui vous protègent contre une mauvaise utilisation des variables locales et des retours `ref` :

- Vous devez ajouter le mot clé `ref` à la signature de méthode et à toutes les instructions `return` dans une méthode.
  - Cela permet de clarifier le retour par référence tout au long de la méthode.
- Un `ref return` peut être assigné à une variable de la valeur ou à une variable `ref`.
  - L'appelant contrôle si la valeur de retour est copiée ou non. L'omission du modificateur `ref` lors de l'assignation de la valeur de retour indique que l'appelant souhaite une copie de la valeur, pas une référence au stockage.
- Vous ne pouvez pas affecter une valeur de retour de méthode standard à une variable locale `ref`.
  - Cela rejette les instructions telles que `ref int i = sequence.Count();`.
- Vous ne pouvez pas retourner un `ref` à une variable dont la durée de vie ne s'étend pas au-delà de l'exécution de la méthode.
  - Cela signifie que vous ne pouvez pas retourner une référence à une variable locale ni une variable avec une étendue similaire.
- Les variables locales et les retours `ref` ne peuvent pas être utilisés avec les méthodes Async.
  - Le compilateur ne peut pas savoir si la variable référencée a été définie à sa valeur finale quand la méthode Async est retournée.

L'ajout de variables locales ref et de retours ref permet d'utiliser des algorithmes qui sont plus efficaces en évitant la copie de valeurs, ou d'effectuer plusieurs fois des opérations de déréférencement.

L'ajout de `ref` à la valeur de retour est une modification compatible avec la source. Le code existant est compilé, mais la valeur de retour référencée est copiée lorsqu'elle est assignée. Les appelants doivent mettre à jour le stockage pour la valeur de retour sur une variable locale `ref` afin de stocker la valeur de retour en tant que référence.

Désormais, les variables locales `ref` peuvent être réassignées pour faire référence à d'autres instances, après avoir été initialisées. Le code suivant effectue maintenant une compilation :

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Pour plus d'informations, consultez l'article sur les `ref` retours et les `ref` variables locales, ainsi que l'article sur `foreach`.

Pour plus d'informations, voir l'article ref, mot clé.

Expressions `ref` conditionnelles

Enfin, l'expression conditionnelle peut produire comme résultat une référence plutôt qu'une valeur. Prenons par exemple la ligne suivante, qui récupère une référence au premier élément dans un des deux tableaux :

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

La variable `r` est une référence à la première valeur de `arr` ou `otherArr`.

Pour plus d'informations, voir Opérateur conditionnel (?:) dans la référence sur le langage.

`in` modificateur de paramètre

Le `in` mot clé complète les mots clés REF et out existants pour passer des arguments par référence. Le mot clé `in` spécifie que l'argument doit être passé par référence, mais la méthode appelée ne modifie pas la valeur.

Vous pouvez déclarer des surcharges qui passent par valeur ou par référence ReadOnly, comme indiqué dans le code suivant :

```
static void M(S arg);
static void M(in S arg);
```

La valeur par valeur (tout d'abord dans l'exemple précédent) est meilleure que la version de référence par ReadOnly. Pour appeler la version avec l'argument de référence en lecture seule, vous devez inclure le modificateur `in` lors de l'appel de la méthode.

Pour plus d'informations, consultez l'article sur le `in` modificateur de paramètre.

D'autres types prennent en charge l'instruction `fixed`

L'instruction `fixed` prenait en charge un ensemble limité de types. À partir de C# 7.3, tout type contenant une méthode `GetPinnableReference()` qui retourne `ref T` ou `ref readonly T` peut être `fixed`. L'ajout de cette fonctionnalité signifie que `fixed` peut être utilisé avec System.Span<T> et des types connexes.

Pour plus d'informations, consultez l'article sur les `fixed` instructions dans la référence du langage.

L'indexation des champs `fixed` ne nécessite aucun épinglage

Prenons le struct suivant :

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

Dans les versions antérieures de C#, vous deviez épingler une variable pour accéder à un des entiers faisant partie de `myFixedField`. À présent, le code suivant se compile sans épingler la variable `p` à l'intérieur d'une instruction `fixed` distincte :

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

La variable `p` accède à un élément dans `myFixedField`. Vous n'avez pas besoin de déclarer une variable `int*` distincte. Vous avez toujours besoin d'un `unsafe` contexte. Dans les versions antérieures de C#, vous devez déclarer un second pointeur fixe :

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

Pour plus d'informations, consultez l'article sur l' `fixed` instruction.

Les tableaux `stackalloc` prennent en charge les initialiseurs

Vous avez été en mesure de spécifier les valeurs des éléments dans un tableau lors de son initialisation :

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Maintenant, la même syntaxe peut être appliquée aux tableaux déclarés avec `stackalloc` :

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

Pour plus d'informations, consultez l'article relatif à l' `stackalloc` opérateur .

Contraintes génériques améliorées

Vous pouvez maintenant spécifier le type System.Enum ou System.Delegate en tant que contraintes de classe de base pour un paramètre de type.

Vous pouvez également utiliser la nouvelle `unmanaged` contrainte pour spécifier qu'un paramètre de type doit être un type non managéqui n'accepte pas les valeurs NULL.

Pour plus d'informations, consultez les articles sur les contraintes et contraintes `where` génériques sur les paramètres de type.

L'ajout de ces contraintes aux types existants est une modification incompatible. Les types génériques fermés peuvent ne plus respecter ces nouvelles contraintes.

Types de retour async généralisés

Le retour d'un objet `Task` à partir de méthodes async peut introduire des goulots d'étranglement au niveau des performances dans certains chemins. `Task` est un type référence. Si vous l'utilisez, vous allouez donc un objet. Dans les cas où une méthode déclarée avec le modificateur `async` retourne un résultat mis en cache, ou si elle s'exécute de manière synchrone, le coût en termes de temps induit par les allocations supplémentaires peut s'avérer significatif dans les sections de code critiques pour les performances. Cela peut devenir coûteux si ces allocations se produisent dans des boucles serrées.

La nouvelle fonctionnalité du langage signifie que les types de retour des méthodes async ne se limitent pas à `Task`, `Task<T>` et `void`. Le type retourné doit toujours correspondre au modèle async, ce qui signifie qu'une méthode `GetAwaiter` doit être accessible. En guise d'exemple concret, le `ValueTask` type a été ajouté à .net pour utiliser cette nouvelle fonctionnalité de langage :

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

> **NOTE**
> Vous devez ajouter le package NuGet `System.Threading.Tasks.Extensions` > pour pouvoir utiliser le ValueTask<TResult> type.

Cette amélioration s'avère particulièrement utile pour les auteurs de bibliothèques afin d'éviter d'allouer un `Task` dans le code critique pour les performances.

Nouvelles options du compilateur

De nouvelles options du compilateur prennent en charge la nouvelle build et les scénarios DevOps pour les programmes C#.

Génération d'assemblys de références

Il existe deux nouvelles options de compilateur qui génèrent des *assemblys de référence uniquement*: -REFOUT et -refonly. Les articles en lien décrivent plus en détail ces options et les assemblys de références.

Signature publique ou Open Source

L'option du compilateur `-publicsign` indique au compilateur de signer l'assembly à l'aide d'une clé publique. L'assembly est marqué comme étant signé, mais la signature provient de la clé publique. Cette option vous permet de générer des assemblys signés à partir de projets open source à l'aide d'une clé publique.

Pour plus d'informations, consultez l'article Option du compilateur -publicsign.

pathmap

L'option du compilateur `-pathmap` indique au compilateur de remplacer les chemins sources de l'environnement de build par des chemins sources mappés. L'option `-pathmap` contrôle le chemin source écrit par le compilateur vers les fichiers PDB ou pour CallerFilePathAttribute.

Pour plus d'informations, consultez l'article Option du compilateur -pathmap.

En savoir plus sur les modifications avec rupture dans le compilateur C#

02/11/2020 • 2 minutes to read • Edit Online

L'équipe Roslyn gère une liste des modifications avec rupture dans les compilateurs C# et Visual Basic. Vous trouverez des informations sur ces modifications à l'adresse suivante sur le référentiel GitHub :

- Modifications avec rupture dans VS2019 version 16,8 qui seront introduites pour .NET 5,0 et C# 9,0
- Modifications avec rupture dans VS2019 Update 1 et autres comparées à VS2019
- Modifications avec rupture depuis VS2017 (C# 7)
- Modifications avec rupture dans Roslyn 3,0 (VS2019) à partir de Roslyn 2. * (VS2017)
- Modifications avec rupture dans Roslyn 2,0 (VS2017) à partir de Roslyn 1. * (VS2015) et du compilateur C# natif (VS2013 et antérieur).
- Modifications avec rupture dans Roslyn 1,0 (VS2015) à partir du compilateur C# natif (VS2013 et Previous).
- Modification de la version Unicode en C# 6

Histoire du langage C#

16/12/2020 • 23 minutes to read • Edit Online

Cet article fournit un historique de chaque version majeure du langage C#. L'équipe C# continue d'innover et d'ajouter de nouvelles fonctionnalités. L'état détaillé des fonctionnalités de langage, notamment celles prises en compte pour les versions à venir, est disponible dans le dépôt dotnet/roslyn sur GitHub.

> **IMPORTANT**
>
> Le langage C# s'appuie sur des types et des méthodes dans ce que la spécification C# définit comme une *bibliothèque standard* pour certaines fonctionnalités. La plateforme .NET fournit ces types et ces méthodes dans des packages. Par exemple, le traitement des exceptions. Chaque instruction ou expression `throw` est vérifiée pour s'assurer de l'objet levé est dérivé de Exception. De même, chaque `catch` est vérifié pour assurer que le type intercepté est dérivé de Exception. Chaque version peut ajouter de nouvelles spécifications. Pour utiliser les dernières fonctionnalités de langage dans les environnements plus anciens, vous devrez peut-être installer des bibliothèques spécifiques. Ces dépendances sont décrites dans la page pour chaque version spécifique. Pour en savoir plus, consultez les relations entre le langage et la bibliothèque pour l'arrière-plan sur cette dépendance.

Les outils de build C# considèrent la dernière version majeure du langage comme la version du langage par défaut. Il peut exister des versions intermédiaires entre les versions majeures, détaillées dans d'autres articles de cette section. Pour utiliser les fonctionnalités les plus récentes dans une version mineure, vous devez configurer la version du langage du compilateur et sélectionner la version. Il y a eu des mises en production à trois points depuis C# 7,0 :

- C# 7,3 :
  - C# 7.3 est disponible à compter de Visual Studio 2017 version 15.7 et du Kit SDK .NET Core 2.1.
- C# 7,2 :
  - C# 7,2 est disponible à partir de Visual Studio 2017 version 15,5 et du Kit de développement logiciel (SDK) .net Core 2,0.
- C# 7,1 :
  - C# 7.1 est disponible à compter de Visual Studio 2017 version 15.3 et du Kit SDK .NET Core 2.0.

C# version 1.0

Lorsque vous vous retrouverez, la version 1,0 de C#, publiée avec Visual Studio .NET 2002, s'est recherchée comme Java. Dans ses objectifs de conception énoncés pour ECMA, il cherchait à être un « langage orienté objet simple, moderne et généraliste ». À l'époque, ressembler à Java signifiait qu'il avait atteint ces objectifs de conception.

Mais si vous repensez à C# 1.0 maintenant, cela peut vous donner le vertige. Il lui manquait des fonctionnalités asynchrones intégrées et certaines des fonctionnalités liées aux génériques qui sont aujourd'hui pour vous une évidence. En fait, il ne proposait pas du tout de génériques. Et LINQ ? Pas encore disponible. Il fallait attendre encore plusieurs années.

C# version 1.0 semblait dénué de fonctionnalités, par rapport à aujourd'hui. Vous vous retrouviez à écrire plus ou moins du code détaillé. Mais il faut bien commencer quelque part. C# version 1.0 était une alternative viable à Java sur la plateforme Windows.

Les principales fonctionnalités du langage C# 1.0 étaient les suivantes :

- Classes

- Structures
- Interfaces
- Événements
- Propriétés
- Délégués
- Opérateurs et expressions
- Instructions
- Attributs

C# version 1.2

La version C# 1,2 est fournie avec Visual Studio .NET 2003. Cette version contenait quelques améliorations mineures du langage. La principale est que, à compter de cette version, le code était généré dans une boucle `foreach` (appelée Dispose) sur un IEnumerator quand ce IEnumerator implémentait IDisposable.

C# version 2.0

Les choses commencent alors à devenir intéressantes. Examinons certaines fonctionnalités majeures de C# 2.0, sorti en 2005, en même temps que Visual Studio 2005 :

- Génériques
- Types partiels
- Méthodes anonymes
- Types valeur Nullable
- Itérateurs
- Covariance et contravariance

D'autres fonctionnalités de C# 2.0 ajoutaient des capacités aux fonctionnalités existantes :

- Accessibilité distincte des accesseurs Get/Set
- Conversions de groupes de méthodes (délégués)
- Classes statiques
- Inférence de délégué

Alors que C# avait démarré en tant que langage orienté objet générique, C# version 2.0 a rapidement changé tout ça. En effet, maintenant que le langage existait, il s'agissait de s'attaquer à certains problèmes majeurs que rencontraient les développeurs. Et de s'y attaquer de manière significative.

Avec les génériques, les types et les méthodes peuvent fonctionner sur un type arbitraire tout en assurant quand même la cohérence des types. À titre d'exemple, un List<T> vous permet d'avoir `List<string>` ou `List<int>` et d'effectuer des opérations qui maintiennent la cohérence des types sur des chaînes ou entiers alors que vous itérez en leur sein. Il vaut mieux utiliser des génériques que créer `ListInt` qui dérive de `ArrayList` ou effectuer un cast de `Object` pour chaque opération.

C# version 2.0 a introduit les itérateurs. En bref, les itérateurs permettent d'examiner tous les éléments dans un `List` (ou d'autres types énumérables) avec une boucle `foreach`. Le fait de disposer d'itérateurs comme composants de premier ordre du langage en a considérablement amélioré la lisibilité et la capacité de raisonnement des utilisateurs vis-à-vis du code.

Pourtant, C# était toujours en train de courir derrière Java. Java avait déjà publié des versions qui incluaient des génériques et des itérateurs. Mais cela allait bientôt changer quand les deux langages poursuivraient des chemins différents.

C# version 3.0

C# version 3.0 est apparu fin 2007, en même temps que Visual Studio 2008, même si l'éventail complet des fonctionnalités du langage n'a réellement voir le jour qu'avec le .NET Framework version 3.5. Cette version a insufflé un changement majeur dans l'évolution de C#. Elle a imposé C# en tant que langage de programmation vraiment formidable. Examinons certaines fonctionnalités importantes dans cette version :

- Propriétés implémentées automatiquement
- Types anonymes
- Expressions de requête
- Expressions lambda
- Arborescences d'expressions
- Méthodes d'extension
- Variables locales implicitement typées
- Méthodes partielles
- Initialiseurs d'objets et de collections

Rétrospectivement, nombre de ces fonctionnalités semblent à la fois inéluctables et inséparables. Elles s'assemblent de façon stratégique. L'expression de requête, également appelée LINQ (Language-Integrated Query), était globalement considérée comme la fonctionnalité remarquable de cette version de C#.

Un point de vue plus nuancé s'intéresse aux arborescences d'expressions, aux expressions lambda et aux types anonymes sur lesquels se construit LINQ. Mais, dans les deux cas, C# 3.0 présentait un concept révolutionnaire. C# 3.0 jetait les bases qui allaient permettre à C# de devenir un langage fonctionnel/orienté objet hybride.

Plus précisément, il devint alors possible d'écrire des requêtes déclaratives de style SQL pour effectuer des opérations sur des collections, entre autres choses. Au lieu d'écrire une boucle `for` pour calculer la moyenne d'une liste d'entiers, vous pouviez alors simplement utiliser `list.Average()`. La combinaison des expressions de requête et des méthodes d'extension a permis à de telles listes d'entiers de paraître beaucoup plus intelligentes.

Il a fallu du temps aux utilisateurs pour comprendre et intégrer ce concept, mais ils y sont progressivement parvenus. Et aujourd'hui, bien des années plus tard, le code est beaucoup plus concis, simple et fonctionnel.

C# version 4.0

La version 4,0 de C#, publiée avec Visual Studio 2010, aurait eu un temps difficile à atteindre l'État révolutionnaire de la version 3,0. Avec la version 3.0, le langage C# est bel et bien sorti de l'ombre de Java pour être propulsé sur le devant de la scène. Il allait même rapidement devenir élégant.

La version suivante introduisit de nouvelles fonctionnalités intéressantes :

- Liaison dynamique
- Arguments nommés/facultatifs
- Covariance et contravariance génériques
- Types interop incorporés

Les types interop incorporés ont permis d'atténuer une difficulté de déploiement. La covariance et la contravariance génériques vous donnent plus de contrôle sur l'utilisation des génériques, mais elles sont un peu académiques et probablement plus appréciées des auteurs de frameworks et de bibliothèques. Les paramètres nommés et facultatifs vous permettent d'éliminer les nombreuses surcharges de méthode et s'avèrent plus pratiques. Mais aucune de ces fonctionnalités ne représente un vrai changement de paradigme.

La fonctionnalité majeure était plutôt l'introduction du mot clé `dynamic`. Le mot clé `dynamic` introduisit dans C# version 4.0 la possibilité de remplacer le compilateur lors de la saisie au moment de la compilation. En utilisant le mot clé dynamic, vous pouvez créer des constructions semblables aux langages dynamiquement typés comme

JavaScript. Vous pouvez créer un `dynamic x = "a string"`, puis y ajouter six, et ainsi laisser le runtime se débrouiller avec ce qui doit se produire par la suite.

La liaison dynamique offre la possibilité de commettre des erreurs, mais également de contrôler davantage le langage.

C# version 5.0

C# version 5,0, fourni avec Visual Studio 2012, était une version ciblée du langage. Presque tous les efforts déployés pour cette version portaient sur un autre concept révolutionnaire du langage : le modèle `async` et `await` pour la programmation asynchrone. Voici la liste des fonctionnalités principales :

- Membres asynchrones
- Attributs d'informations de l'appelant

Voir aussi

- Projet de code : Attributs des informations de l'appelant en C# 5.0

L'attribut d'informations de l'appelant vous permet de récupérer facilement des informations sur le contexte d'exécution sans avoir recours à une multitude de code de réflexion réutilisable. Nombre de ses usages ont trait aux diagnostics et aux tâches de journalisation.

Mais `async` et `await` sont les véritables vedettes de cette version. Quand ces fonctionnalités sont apparues en 2012, C# a encore modifié la donne en intégrant l'asynchronie en tant que participant de premier ordre dans le langage. Si vous avez déjà eu affaire à de longues opérations et à l'implémentation de rappels, vous avez probablement adoré cette fonctionnalité de langage.

C# version 6.0

Avec les versions 3.0 et 5.0, C# avait ajouté d'importantes nouvelles fonctionnalités à un langage orienté objet. Avec la version 6,0, publiée avec Visual Studio 2015, il ne s'agissait pas d'une fonctionnalité de déploiement dominant et de libérer à la place de nombreuses fonctionnalités plus petites qui rendaient la programmation C# plus productive. En voici quelques-unes :

- Importations statiques
- Filtres d'exceptions
- Initialiseurs de propriétés automatiques
- Membres expression-bodied
- Propagateur Null
- Interpolation de chaîne
- opérateur nameof

Quelques autres nouvelles fonctions :

- Initialiseurs d'index
- Await dans des blocs catch/finally
- Valeurs par défaut pour les propriétés d'accesseur Get

Chacune de ces fonctionnalités est intéressante individuellement. Mais si vous les examinez dans leur ensemble, un modèle intéressant se dégage. Dans cette version, C# a éliminé le texte réutilisable du langage pour rendre le code plus laconique et plus lisible. Ainsi, pour les amateurs de code propre et simple, cette version du langage était une énorme victoire.

Une autre nouveauté a été proposée avec cette version, même s'il ne s'agit pas d'une fonctionnalité de langage traditionnelle à proprement dit. Le compilateur Roslyn a été publié en tant que service. Le compilateur C# est à

présent écrit en C# et vous pouvez l'utiliser dans le cadre de vos efforts de programmation.

C# version 7.0

C# version 7,0 a été publié avec Visual Studio 2017. Cette version propose des évolutions intéressantes dans l'esprit de C# 6.0, mais sans le compilateur en tant que service. Voici quelques-unes des nouvelles fonctionnalités :

- Variables out
- Tuples et déconstruction
- Critères spéciaux
- Fonctions locales
- Membres expression-bodied étendus
- Variables locales et retours ref

Autres fonctionnalités disponibles :

- Éléments ignorés
- Littéraux binaires et séparateurs numériques
- Expressions throw

Toutes ces fonctionnalités offrent de nouvelles capacités appréciables aux développeurs ainsi que la possibilité d'écrire du code encore plus propre. Il s'agit notamment de condenser la déclaration des variables à utiliser avec le mot clé `out` et en autorisant plusieurs valeurs de retour par le biais d'un tuple.

Par ailleurs, les utilisations de C# sont de plus en plus larges. .NET Core cible à présent n'importe quel système d'exploitation et garde les yeux rivés sur le cloud et la portabilité. Ces nouvelles fonctions occupent sans aucun doute les pensées des concepteurs du langage, en plus des fonctionnalités à venir.

Version C# 7,1

C# a démarré la publication de *versions point* avec c# 7,1. Cette version a ajouté l'élément de configuration de la sélection de version de langage , trois nouvelles fonctionnalités de langage et un nouveau comportement de compilateur.

Les nouvelles fonctionnalités de langage de cette version sont :

- `async``Main` méthode
  - Le point d'entrée pour une application peut avoir le modificateur `async`.
- `default` expressions littérales
  - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Noms des éléments de tuple inférés
  - Les noms des éléments de tuple peuvent être inférés dans de nombreux cas à partir de l'initialisation du tuple.
- Critères spéciaux sur les paramètres de type générique
  - Il est possible d'utiliser des expressions de critères spéciaux sur les variables dont le type est un paramètre de type générique.

Enfin, le compilateur a deux options, `-refout` et `-refonly`, qui contrôlent la génération d'assemblys de références.

Version C# 7,2

C# 7,2 a ajouté plusieurs fonctionnalités de langage de petite taille :

- Techniques d'écriture de code safe et efficace
  - Une combinaison des améliorations de la syntaxe qui permettent d'utiliser les types valeur avec la sémantique de référence.
- Arguments nommés non placés en position de fin
  - Les arguments nommés peuvent être suivis par des arguments de position.
- Traits de soulignement de début dans les littéraux numériques
  - Les littéraux numériques peuvent maintenant comporter des traits de soulignement de début avant tout chiffre affiché.
- `private protected` modificateur d'accès
  - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Expressions conditionnelles `ref`
  - Le résultat d'une expression conditionnelle (`?:`) peut maintenant être une référence.

Version C# 7,3

Il existe deux thèmes principaux pour la version C# 7.3. Un thème fournit des fonctionnalités permettant au code sécurisé d'être aussi performant que le code non sécurisé. Le second thème fournit des améliorations incrémentielles aux fonctionnalités existantes. En outre, de nouvelles options de compilateur ont été ajoutées à cette version.

Les nouvelles fonctionnalités suivantes prennent en charge le thème de meilleures performances pour le code sécurisé :

- Vous pouvez accéder à des champs fixes sans épinglage.
- Vous pouvez réassigner des `ref` variables locales.
- Vous pouvez utiliser des initialiseurs sur des `stackalloc` tableaux.
- Vous pouvez utiliser des `fixed` instructions avec n'importe quel type prenant en charge un modèle.
- Vous pouvez utiliser des contraintes génériques supplémentaires.

Les améliorations suivantes ont été apportées aux fonctionnalités existantes :

- Vous pouvez tester `==` et `!=` avec des types de tuple.
- Vous pouvez utiliser des variables d'expression dans d'autres emplacements.
- Vous pouvez joindre des attributs à un champ de stockage de propriétés implémentées automatiquement.
- La résolution de la méthode lorsque des arguments diffèrent de `in` a été améliorée.
- La résolution de la surcharge comporte maintenant moins de cas ambigus.

Les nouvelles options du compilateur sont les suivantes :

- `-publicsign` pour activer la signature d'assemblys Open Source Software (OSS).
- `-pathmap` pour fournir un mappage des répertoires sources.

Version C# 8,0

C# 8,0 est la première version majeure de C# qui cible spécifiquement .NET Core. Certaines fonctionnalités s'appuient sur les nouvelles fonctionnalités CLR, d'autres sur les types de bibliothèque ajoutés uniquement dans .NET Core. C# 8,0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Membres ReadOnly
- Méthodes d'interface par défaut
- Améliorations des critères spéciaux:

    - Expressions de commutateur
    - Modèles de propriétés
    - Modèles de tuples
    - Modèles positionnels
- Utilisation des déclarations
- Fonctions locales statiques
- Structs ref jetables
- Types références Nullables
- Flux asynchrones
- Index et plages
- Assignation de fusion Null
- Types construits non managés
- Stackalloc dans les expressions imbriquées
- Amélioration des chaînes textuelles interpolées

Les membres d'interface par défaut nécessitent des améliorations dans le CLR. Ces fonctionnalités ont été ajoutées dans le CLR pour .NET Core 3,0. Les plages et les index, et les flux asynchrones requièrent de nouveaux types dans les bibliothèques .NET Core 3,0. Les types de référence Nullable, bien qu'implémentés dans le compilateur, sont bien plus utiles lorsque les bibliothèques sont annotées pour fournir des informations sémantiques sur l'État null des arguments et les valeurs de retour. Ces annotations sont ajoutées dans les bibliothèques .NET Core.

Article initialement publié sur le blog NDepend, avec l'aimable autorisation d'Erik Dietrich et de Patrick Smacchia.

Relations entre les fonctionnalités de langage et les types de bibliothèque

02/11/2020 • 4 minutes to read • Edit Online

La définition du langage C# nécessite une bibliothèque standard avec certains types et certains membres accessibles sur ces types. Le compilateur génère du code qui a besoin d'utiliser ces types et membres pour de nombreuses fonctionnalités de langage différentes. Quand cela est nécessaire, il existe des packages NuGet qui contiennent tous les types à utiliser pour des versions plus récentes du langage si vous écrivez du code destiné à des environnements où ces types ou membres n'ont pas encore été déployés.

Cette dépendance sur les fonctionnalités de la bibliothèque standard fait partie du langage C# depuis le début. Dans la première version, il y avait les types suivants :

- Exception : utilisé pour toutes les exceptions générées par le compilateur.
- String : le type `string` C# est un synonyme de String.
- Int32 : synonyme de `int`.

Cette première version était simple : le compilateur et la bibliothèque standard étaient fournis ensemble, et il n'y avait qu'une seule version de chacun.

Dans les versions ultérieures de C#, de nouveaux types ou membres ont parfois été ajoutés aux dépendances. Par exemple, INotifyCompletion, CallerFilePathAttribute et CallerMemberNameAttribute. C# 7.0 poursuit dans ce sens en ajoutant une dépendance sur ValueTuple pour implémenter la fonctionnalité de langage tuples.

L'équipe de conception du langage s'efforce de réduire la surface d'exposition des types et des membres requis dans une bibliothèque standard conforme. Cet objectif s'ajoute à la nécessité de conserver une conception épurée à mesure que de nouvelles fonctionnalités de bibliothèque sont incorporées au langage. De nouvelles fonctionnalités seront ajoutées dans les futures versions du langage C# qui nécessiteront de nouveaux types et membres dans une bibliothèque standard. Il est donc important de bien comprendre comment gérer ces dépendances dans votre travail.

Gestion des dépendances

Les outils du compilateur C# sont maintenant dissociés du cycle de mise en production des bibliothèques .NET sur les plateformes prises en charge. En effet, les bibliothèques .NET n'ont pas toutes les mêmes cycles de mise en production : le .NET Framework sur Windows est fourni en tant que mise à jour Windows Update, .NET Core est fourni selon une planification distincte, et les versions Xamarin des mises à jour de bibliothèque sont fournies avec les outils Xamarin pour chaque plateforme cible.

La plupart du temps, ces changements passeront inaperçus pour les utilisateurs. Toutefois, si vous utilisez une version du langage plus récente qui fait appel à des fonctionnalités actuellement non disponibles dans les bibliothèques .NET sur cette plateforme, vous devez référencer les packages NuGet pour fournir ces nouveaux types nécessaires. Une fois que les plateformes prises en charge par votre application auront été mises à jour avec les nouvelles versions de framework, vous pourrez supprimer la référence ajoutée.

Grâce à cette dissociation, vous pouvez utiliser les nouvelles fonctionnalités de langage même quand vous ciblez des machines qui n'ont pas le framework approprié.

Considérations relatives à la version et la mise à jour pour les développeurs C#

02/11/2020 • 4 minutes to read • Edit Online

La compatibilité est un objectif très important alors que des nouvelles fonctionnalités sont ajoutées au langage C#. Dans presque tous les cas, le code existant peut être recompilé avec une nouvelle version du compilateur sans problème.

Vous devrez peut-être faire plus attention lorsque vous adoptez de nouvelles fonctionnalités de langage dans une bibliothèque. Vous créez peut-être une nouvelle bibliothèque avec des fonctionnalités incluses dans la version la plus récente et avez besoin de vous assurer que les applications créées à l'aide de versions précédentes du compilateur peuvent l'utiliser. Ou vous mettez peut-être à niveau une bibliothèque existante et la plupart de vos utilisateurs ne disposent pas encore des versions mises à niveau. Lorsque vous prenez des décisions sur l'adoption de nouvelles fonctionnalités, vous devez prendre en compte deux variantes de la compatibilité : compatibilité avec la source et compatibilité binaire.

Modifications compatibles binaires

Les modifications apportées à votre bibliothèque sont **compatibles binaires** lorsque votre bibliothèque mise à jour peut être utilisée sans régénération des applications et des bibliothèques qui l'utilisent. Les assemblys dépendants n'ont pas besoin d'être reconstruits et aucune modification du code source n'est requise. Les modifications compatibles binaires sont également des modifications compatibles avec la source.

Modifications compatibles avec la source

Les modifications apportées à votre bibliothèque sont **compatibles avec la source** lorsque les applications et les bibliothèques qui utilisent votre bibliothèque ne nécessitent pas de modifications du code source, mais que la source doit être recompilée avec la nouvelle version pour fonctionner correctement.

Modifications incompatibles

Si une modification n'est ni **compatible avec la source** ni **compatible binaire**, des modifications du code source, ainsi qu'une recompilation sont requises dans les applications et les bibliothèques dépendantes.

Évaluer votre bibliothèque

Ces concepts de compatibilité affectent les déclarations publiques et protégées de votre bibliothèque, pas son implémentation interne. L'adoption de nouvelles fonctionnalités en interne est toujours **compatible binaire**.

Les modifications **compatibles binaires** fournissent une nouvelle syntaxe qui génère le même code compilé pour les déclarations publiques que l'ancienne syntaxe. Par exemple, la modification d'une méthode en un membre expression-bodied est une modification **compatible binaire** :

Code d'origine :

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

Nouveau code :

```
public double CalculateSquare(double value) => value * value;
```

Les modifications **compatibles avec la source** introduisent une syntaxe qui modifie le code compilé pour un membre public, mais de manière compatible avec les sites d'appel existants. Par exemple, la modification d'une signature de méthode à partir d'un paramètre de valeur by en un paramètre de référence by `in` est compatible avec la source mais pas compatible binaire :

Code d'origine :

```
public double CalculateSquare(double value) => value * value;
```

Nouveau code :

```
public double CalculateSquare(in double value) => value * value;
```

Les articles Nouveautés indiquent si l'introduction d'une fonctionnalité qui affecte les déclarations publiques est compatible avec la source ou compatible binaire.

Types (Guide de programmation C#)

16/12/2020 • 25 minutes to read • Edit Online

Types, variables et valeurs

C# est un langage fortement typé. Chaque variable et chaque constante ont un type, tout comme chaque expression qui fournit une valeur. Chaque déclaration de méthode spécifie un nom, un nombre de paramètres, ainsi que le type et le type (valeur, référence ou sortie) pour chaque paramètre d'entrée et pour la valeur de retour. La bibliothèque de classes .NET définit un ensemble de types numériques intégrés et des types plus complexes qui représentent un large éventail de constructions logiques, telles que le système de fichiers, les connexions réseau, les collections et les tableaux d'objets, ainsi que les dates. Un programme C# classique utilise des types de la bibliothèque de classes et des types définis par l'utilisateur qui modélisent les concepts propres au domaine du problème du programme.

Les informations stockées dans un type peuvent inclure les éléments suivants :

- Espace de stockage nécessaire pour une variable du type.
- Valeurs minimale et maximale que le type peut représenter.
- Membres (méthodes, champs, événements, etc.) que le type contient.
- Type de base dont le type est hérité.
- Interface (s) qu'il implémente.
- Emplacement où sera allouée la mémoire pour les variables au moment de l'exécution.
- Sortes d'opérations autorisées.

Le compilateur utilise les informations de type pour s'assurer que toutes les opérations effectuées dans votre code sont de *type sécurisé*. Par exemple, si vous déclarez une variable de type int, le compilateur vous permet d'utiliser la variable dans des opérations d'addition et de soustraction. Si vous essayez d'effectuer ces mêmes opérations avec une variable de type bool, le compilateur génère une erreur, comme illustré dans l'exemple suivant :

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> Développeurs C et C++ : notez que dans C#, bool n'est pas convertible en int.

Le compilateur incorpore les informations de type dans le fichier exécutable sous forme de métadonnées. Le common language runtime (CLR) utilise ces métadonnées au moment de l'exécution pour garantir que le type est sécurisé lorsqu'il alloue et libère de la mémoire.

Spécification de types dans les déclarations de variable

Lorsque vous déclarez une variable ou une constante dans un programme, vous devez spécifier son type ou utiliser le mot clé var pour permettre au compilateur de déduire le type. L'exemple suivant montre des déclarations de variable qui utilisent des types numériques intégrés et des types complexes définis par l'utilisateur :

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Les types de paramètres de méthode et de valeurs de retour sont spécifiés dans la déclaration de méthode. La signature suivante présente une méthode qui requiert int comme argument d'entrée et retourne une chaîne :

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Une fois que vous avez déclaré une variable, vous ne pouvez pas la redéclarer avec un nouveau type et vous ne pouvez pas assigner une valeur qui n'est pas compatible avec son type déclaré. Par exemple, vous ne pouvez pas déclarer un int et lui assigner une valeur booléenne `true` . Toutefois, les valeurs peuvent être converties en d'autres types, par exemple quand elles sont assignées à de nouvelles variables ou passées en tant qu'arguments de méthode. Une *conversion de type* qui n'entraîne pas de perte de données est effectuée automatiquement par le compilateur. Une conversion susceptible de causer la perte de données exige une variable *cast* dans le code source.

Pour plus d'informations, consultez Cast et conversions de types.

Types intégrés

C# fournit un jeu standard de types intégrés pour représenter les entiers, les valeurs à virgule flottante, les expressions booléennes, les caractères de texte, les valeurs décimales et d'autres types de données. Il existe également des types `string` et `object` intégrés. Ces types peuvent être utilisés dans n'importe quel programme C#. Pour obtenir la liste complète des types intégrés, consultez types intégrés.

Types personnalisés

Vous utilisez les constructions struct, classe, interface et enum pour créer vos propres types personnalisés. La bibliothèque de classes .NET est une collection de types personnalisés fournie par Microsoft que vous pouvez utiliser dans vos propres applications. Par défaut, les types les plus fréquemment utilisés dans la bibliothèque de classes sont disponibles dans tous les programmes C#. D'autres deviennent disponibles uniquement lorsque vous ajoutez explicitement une référence de projet à l'assembly dans lequel elles sont définies. Une fois que le compilateur a une référence à l'assembly, vous pouvez déclarer des variables (et des constantes) des types déclarés dans cet assembly dans le code source. Pour plus d'informations, consultez Bibliothèque de classes .NET.

Système de type commun

Il est important de comprendre deux points fondamentaux concernant le système de type dans .NET :

- Il prend en charge le principe d'héritage. Les types peuvent dériver d'autres types, appelés *types de base*. Le type

dérivé hérite (avec certaines restrictions) des méthodes, des propriétés et des autres membres du type de base. Le type de base peut à son tour dériver d'un autre type, auquel cas le type dérivé hérite des membres des deux types de base dans sa hiérarchie d'héritage. Tous les types, notamment les types numériques intégrés comme System.Int32 (mot clé C# : int), dérivent au final d'un seul type de base, qui est System.Object (mot clé C# : object). Cette hiérarchie de types unifiée est appelée Système de type commun (CTS). Pour plus d'informations sur l'héritage dans C#, consultez Héritage.

- Chaque type du CTS est défini comme *type valeur* ou *type référence*. Ces types incluent tous les types personnalisés dans la bibliothèque de classes .NET et également vos propres types définis par l'utilisateur. Les types que vous définissez à l'aide du mot clé struct sont des types valeur ; tous les types numériques intégrés sont `structs`. Les types que vous définissez à l'aide du mot clé class sont des types référence. Les types référence et les types valeur ne suivent pas les mêmes règles de compilation et ont un comportement différent au moment de l'exécution.

L'illustration suivante montre la relation entre les types valeur et les types référence dans le CTS.

L'image suivante montre les types valeur et les types référence dans CTS :

> **NOTE**
>
> Vous pouvez voir que les types couramment utilisés sont tous organisés dans l'espace de noms System. Toutefois, l'espace de noms qui contient un type n'a aucune incidence sur le fait qu'il s'agisse d'un type valeur ou d'un type référence.

Types de valeur

Les types valeur dérivent de System.ValueType, qui dérive de System.Object. Les types qui dérivent de System.ValueType ont un comportement spécial dans le CLR. Les variables de type valeur contiennent directement leurs valeurs, ce qui signifie que la mémoire est allouée inline dans le contexte où la variable est déclarée. Il n'existe pas d'allocation de tas ou de charge de garbage collection distincte pour les variables de type valeur.

Il existe deux catégories de types valeur : struct et enum.

Les types numériques intégrés sont des structs et ont des champs et des méthodes auxquels vous pouvez accéder :

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Mais vous les déclarez et leur assignez des valeurs comme s'il s'agissait de types non agrégés simples :

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Les types valeur sont *sealed*, ce qui signifie que vous ne pouvez pas dériver un type de n'importe quel type valeur, par exemple System.Int32 . Vous ne pouvez pas définir un struct pour hériter d'une classe ou d'un struct défini par l'utilisateur, car un struct peut uniquement hériter de System.ValueType . Toutefois, un struct peut implémenter une ou plusieurs interfaces. Vous pouvez effectuer un cast d'un type struct en n'importe quel type d'interface qu'il implémente ; ce Cast entraîne une opération de *boxing* pour encapsuler le struct à l'intérieur d'un objet de type référence sur le tas managé. Les opérations de boxing se produisent quand vous passez un type valeur à une méthode qui prend un System.Object ou tout type d'interface comme paramètre d'entrée. Pour plus d'informations, consultez Conversion boxing et unboxing.

Vous utilisez le mot clé struct pour créer vos propres types valeur personnalisés. En règle générale, un struct est utilisé comme conteneur pour un petit jeu de variables connexes, comme le montre l'exemple suivant :

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Pour plus d'informations sur les structs, consultez types structure. Pour plus d'informations sur les types valeur, consultez types valeur.

L'autre catégorie de types valeur est enum. Un enum définit un jeu de constantes intégrales nommées. Par exemple l'énumération System.IO.FileMode dans la bibliothèque de classes du .NET contient un ensemble d'entiers constants nommés qui spécifient comment un fichier doit être ouvert. Il est défini comme indiqué dans l'exemple suivant :

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

La constante `System.IO.FileMode.Create` a la valeur 2. Toutefois, le nom est bien plus explicite pour les êtres humains qui lisent le code source. pour cette raison, il est préférable d'utiliser des énumérations au lieu de nombres littéraux constants. Pour plus d'informations, consultez System.IO.FileMode.

Toutes les énumérations héritent de System.Enum, qui hérite de System.ValueType. Toutes les règles qui s'appliquent aux structs s'appliquent également aux enums. Pour plus d'informations sur les enums, consultez types énumération.

Types référence

Un type qui est défini comme une classe, un délégué, un tableau ou une interface est un *type référence*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur null tant que vous n'avez pas explicitement créé un objet à l'aide de l'opérateur new ou que vous ne lui avez pas assigné un objet créé

ailleurs à l'aide de `new`, comme indiqué dans l'exemple suivant :

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

Une interface doit être initialisée avec un objet de classe qui l'implémente. Si `MyClass` implémente `IMyInterface`, vous créez une instance de `IMyInterface` comme indiqué dans l'exemple suivant :

```
IMyInterface iface = new MyClass();
```

Quand l'objet est créé, la mémoire est allouée sur le tas managé et la variable contient uniquement une référence à l'emplacement de l'objet. Les types sur le tas managé nécessitent une surcharge lorsqu'ils sont alloués et lorsqu'ils sont récupérés par la fonctionnalité de gestion automatique de la mémoire du CLR, appelée *garbage collection*. Toutefois, garbage collection est également hautement optimisé et, dans la plupart des scénarios, il ne crée pas de problème de performances. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire.

Tous les tableaux sont des types référence, même si leurs éléments sont des types valeur. Les tableaux dérivent implicitement de la classe System.Array, mais vous les déclarez et vous les utilisez avec la syntaxe simplifiée fournie par C#, comme illustré dans l'exemple suivant :

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Les types référence prennent en charge l'héritage. Lorsque vous créez une classe, vous pouvez hériter de toute autre interface ou classe qui n'est pas définie comme sealed, et d'autres classes peuvent hériter de votre classe et substituer vos méthodes virtuelles. Pour plus d'informations sur la création de vos propres classes, consultez Classes et structures. Pour plus d'informations sur l'héritage et les méthodes virtuelles, consultez Héritage.

Types de valeurs littérales

Dans C#, les valeurs littérales reçoivent un type du compilateur. Vous pouvez spécifier la façon dont un littéral numérique doit être typé en ajoutant une lettre à la fin du nombre. Par exemple, pour spécifier que la valeur 4,56 doit être traitée comme une valeur float, ajoutez « f » ou « F » après le nombre : `4.56f`. Si aucune lettre n'est ajoutée, le compilateur déduit un type pour le littéral. Pour plus d'informations sur les types qui peuvent être spécifiés avec des suffixes de lettres, consultez types numériques intégraux et types numériques à virgule flottante.

Étant donné que les littéraux sont typés et que tous les types dérivent en fin de compte System.Object , vous pouvez écrire et compiler du code tel que le code suivant :

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Types génériques

Un type peut être déclaré avec un ou plusieurs *paramètres de type* qui servent d'espace réservé pour le type réel (le

type concret) que le code client fournit lorsqu'il crée une instance du type. Ces types sont appelés *types génériques*. Par exemple, le type .NET System.Collections.Generic.List<T> a un paramètre de type qui, par Convention, porte le nom *T*. Lorsque vous créez une instance du type, vous spécifiez le type des objets que la liste doit contenir, par exemple String :

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

L'utilisation du paramètre de type rend possible la réutilisation de la même classe pour contenir tout type d'élément, sans avoir à convertir chaque élément en object. Les classes de collection génériques sont appelées *collections fortement typées* , car le compilateur connaît le type spécifique des éléments de la collection et peut déclencher une erreur au moment de la compilation si, par exemple, vous tentez d'ajouter un entier à l' `stringList` objet dans l'exemple précédent. Pour plus d'informations, consultez Génériques.

Types implicites, types anonymes et types valeur Nullable

Comme indiqué précédemment, vous pouvez attribuer implicitement un type à une variable locale (mais pas les membres de la classe) à l'aide du mot clé var. La variable reçoit toujours un type au moment de la compilation, mais le type est fourni par le compilateur. Pour plus d'informations, consultez Variables locales implicitement typées.

Il peut être difficile de créer un type nommé pour des ensembles simples de valeurs associées que vous ne souhaitez pas stocker ou passer en dehors des limites de la méthode. Vous pouvez alors créer des *types anonymes*. Pour plus d'informations, consultez types anonymes.

Les types valeur ordinaires ne peuvent pas avoir une valeur null. Toutefois, vous pouvez créer des types valeur Nullable en ajoutant un `?` après le type. Par exemple, `int?` est un type `int` qui peut également avoir la valeur null. Les types valeur Nullable sont des instances du type struct générique System.Nullable<T> . Les types valeur Nullable sont particulièrement utiles lorsque vous passez des données vers et à partir de bases de données dans lesquelles les valeurs numériques peuvent être null. Pour plus d'informations, consultez types valeur Nullable.

Type au moment de la compilation et type d'exécution

Une variable peut avoir différents types au moment de la compilation et au moment de l'exécution. Le *type au moment* de la compilation est le type déclaré ou déduit de la variable dans le code source. Le *type au moment* de l'exécution est le type de l'instance référencée par cette variable. Ces deux types sont souvent les mêmes, comme dans l'exemple suivant :

```
string message = "This is a string of characters";
```

Dans d'autres cas, le type au moment de la compilation est différent, comme indiqué dans les deux exemples suivants :

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

Dans les deux exemples précédents, le type au moment de l'exécution est un `string` . Le type au moment de la compilation est `object` sur la première ligne et `IEnumerable<char>` dans la seconde.

Si les deux types sont différents pour une variable, il est important de comprendre quand le type au moment de la compilation et le type au moment de l'exécution s'appliquent. Le type au moment de la compilation détermine toutes les actions effectuées par le compilateur. Ces actions du compilateur incluent la résolution de l'appel de

méthode, la résolution de surcharge et les casts implicites et explicites disponibles. Le type au moment de l'exécution détermine toutes les actions qui sont résolues au moment de l'exécution. Ces actions au moment de l'exécution incluent la répartition des appels de méthode virtuelle, l'évaluation `is` et les `switch` expressions, ainsi que d'autres API de test de type. Pour mieux comprendre comment votre code interagit avec les types, identifiez l'action qui s'applique à ce type.

Sections connexes

Pour plus d'informations, consultez les articles suivants :

- Cast et conversions de types
- Conversion boxing et unboxing
- Utilisation du type dynamic
- Types valeur
- Types référence
- Classes et structs
- Types anonymes
- Génériques

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Conversion des types de données XML
- Types intégraux

Types références Nullables

16/12/2020 • 18 minutes to read • Edit Online

C# 8.0 introduit des **types référence nullables** et des **types référence non nullables** qui vous permettent d'établir des instructions importantes sur les propriétés de variables de type référence :

- **Une référence ne doit pas être null**. Lorsque les variables ne sont pas censées être null, le compilateur applique des règles qui garantissent que le déréférencement de ces variables est sécurisé sans avoir d'abord vérifié qu'elle n'est pas NULL :
  - La variable doit être initialisée avec une valeur non null.
  - La variable ne peut jamais se voir attribuer la valeur `null`.
- **Une référence peut être null**. Lorsque les variables peuvent être null, le compilateur applique des règles différentes pour garantir que vous avez correctement vérifié une référence null :
  - La variable peut être déréférencée seulement si le compilateur peut garantir que la valeur n'est pas null.
  - Ces variables peuvent être initialisées avec la valeur `null` par défaut et peuvent se voir attribuer la valeur `null` dans un autre code.

Cette nouvelle fonctionnalité offre des avantages significatifs par rapport à la gestion des variables de référence dans les versions antérieures de C#, où l'intention de conception ne peut pas être déterminée à partir de la déclaration de la variable. Le compilateur ne protégeait pas des exceptions de référence null pour les types référence :

- **Une référence peut être null**. Le compilateur n'émet pas d'avertissements quand une variable de type référence est initialisée à `null` ou ultérieurement assignée `null` . Le compilateur émet des avertissements lorsque ces variables sont déréférencées sans contrôles null.
- **Une référence est censée ne pas être null**. Le compilateur ne génère aucun avertissement quand les types référence sont déréférencés. Le compilateur émet des avertissements si une variable est définie sur une expression qui peut être null.

Ces avertissements sont émis au moment de la compilation. Le compilateur n'ajoute pas de contrôles null ou autres constructions d'exécution dans un contexte Nullable. Au moment de l'exécution, une référence Nullable et une référence non Nullable sont équivalentes.

En ajoutant des types référence nullables, vous pouvez déclarer votre intention plus clairement. La valeur `null` est un bon moyen de représenter qu'une variable ne fait pas référence à une valeur. N'utilisez pas cette fonctionnalité pour supprimer toutes les valeurs `null` de votre code. Déclarez plutôt votre intention au compilateur et aux autres développeurs qui lisent votre code. Quand vous déclarez votre intention, le compilateur vous informe si vous écrivez du code qui n'est pas en phase avec cette intention.

Un **type de référence nullable** est inidqué avec la même syntaxe que celle des types valeur nullable : un `?` est ajouté au type de la variable. Par exemple, la déclaration de variable suivante représente une variable de chaîne nullable, `name` :

```
string? name;
```

Toute variable où `?` n'est pas ajouté au nom de type est un **type de référence non Nullable**. Cela comprend toutes les variables de type référence dans le code existant lorsque vous avez activé cette fonctionnalité.

Le compilateur utilise l'analyse statique pour déterminer si une référence nullable est connue pour être non null. Le compilateur vous avertit si vous déréférencez une référence nullable quand elle est susceptible d'être null.

Vous pouvez remplacer ce comportement à l'aide de l' opérateur null-indulgent avec à la `!` suite d'un nom de variable. Par exemple, si vous savez que la variable `name` n'est pas null alors que le compilateur génère un avertissement, vous pouvez écrire le code suivant pour remplacer l'analyse du compilateur :

```
name!.Length;
```

Nullabilité des types

Tout type référence peut avoir l'une des quatre *nullabilités*, qui décrit quand des avertissements sont générés :

- Non *null*: aucune valeur NULL ne peut être assignée aux variables de ce type. Les variables de ce type n'ont pas besoin d'être vérifiées pour savoir si elles ont la valeur null avant d'être déréférencées.
- *Nullable*: la valeur NULL peut être assignée aux variables de ce type. Le déréférencement des variables de ce type sans vérifier `null` au préalable génère un avertissement.
- *Oublie*: oublie est l'état pré-C # 8,0. Les variables de ce type peuvent être déréférencées ou attribuées sans avertissement.
- *Inconnu*: inconnu est généralement utilisé pour les paramètres de type où les contraintes n'indiquent pas au compilateur que le type doit être *Nullable* ou ne peut pas être *null*.

La nullabilité d'un type dans une déclaration de variable est contrôlée par le *contexte nullable* dans lequel la variable est déclarée.

Contextes nullables

Les contextes nullables permettent de contrôler précisément comment le compilateur interprète les variables de type référence. Le **contexte d'annotation Nullable** d'une ligne source donnée est activé ou désactivé. Vous pouvez considérer le compilateur pre-C # 8,0 comme en compilant tout votre code dans un contexte Nullable désactivé : tout type de référence peut être null. Le **contexte d'avertissements Nullable** peut également être activé ou désactivé. Le contexte d'avertissements nullable spécifie les avertissements générés par le compilateur à l'aide de son analyse de flux.

Le contexte d'annotation Nullable et le contexte d'avertissement Nullable peuvent être définis pour un projet à l'aide de l' `Nullable` élément dans votre fichier *.csproj* . Cet élément configure comment le compilateur interprète la nullabilité des types et quels avertissements sont générés. Les paramètres valides sont :

- `enable`: Le contexte d'annotation Nullable est **activé**. Le contexte d'avertissement nullable est **activé**.
  - Les variables d'un type référence, `string` par exemple, sont non nullables. Tous les avertissements de nullabilité sont activés.
- `warnings`: Le contexte d'annotation Nullable est **désactivé**. Le contexte d'avertissement nullable est **activé**.
  - Les variables d'un type référence sont oblivious. Tous les avertissements de nullabilité sont activés.
- `annotations`: Le contexte d'annotation Nullable est **activé**. Le contexte d'avertissement nullable est **désactivé**.
  - Les variables d'un type référence, String par exemple, ne sont pas nullables. Tous les avertissements de nullabilité sont désactivés.
- `disable`: Le contexte d'annotation Nullable est **désactivé**. Le contexte d'avertissement nullable est **désactivé**.
  - Les variables d'un type référence sont oblivious, comme dans les versions antérieures de C#. Tous les avertissements de nullabilité sont désactivés.

Exemple :

```
<Nullable>enable</Nullable>
```

Vous pouvez aussi utiliser des directives pour définir ces mêmes contextes partout dans votre projet :

- `#nullable enable` : Définit le contexte d'annotation Nullable et le contexte d'avertissement Nullable sur **activé**.
- `#nullable disable` : Définit le contexte d'annotation Nullable et le contexte d'avertissement Nullable sur **désactivé**.
- `#nullable restore` : Restaure le contexte d'annotation Nullable et le contexte d'avertissement Nullable aux paramètres du projet.
- `#nullable disable warnings` : Affectez la valeur **désactivé** au contexte d'avertissement Nullable.
- `#nullable enable warnings` : Affectez la valeur **activé** au contexte d'avertissement Nullable.
- `#nullable restore warnings` : Restaure le contexte d'avertissement Nullable aux paramètres du projet.
- `#nullable disable annotations` : Affectez la valeur **désactivé** au contexte d'annotation Nullable.
- `#nullable enable annotations` : Affectez la valeur **activé** au contexte d'annotation Nullable.
- `#nullable restore annotations` : Restaure le contexte d'avertissement d'annotation aux paramètres du projet.

IMPORTANT

Le contexte Nullable global ne s'applique pas aux fichiers de code générés. Dans l'une ou l'autre stratégie, le contexte Nullable est *désactivé* pour tout fichier source marqué comme généré. Cela signifie que toutes les API dans les fichiers générés ne sont pas annotées. Un fichier est marqué comme généré de quatre façons :

1. Dans le fichier. editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
2. Placez `<auto-generated>` ou `<auto-generated/>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.
3. Démarrer le nom de fichier avec *TemporaryGeneratedFile_*
4. Terminez le nom de fichier avec *. Designer.cs*, *. generated.cs*, *. g.cs* ou *. g.i.cs*.

Les générateurs peuvent s'abonner à l'aide de la `#nullable` directive de préprocesseur.

Par défaut, l'annotation Nullable et les contextes d'avertissement sont **désactivés**, y compris les nouveaux projets. Cela signifie que votre code existant est compilé sans modification et sans générer de nouveaux avertissements.

Ces options fournissent deux stratégies distinctes pour mettre à jour une base de code existante afin d'utiliser des types de référence Nullable.

Contexte d'annotation nullable

Le compilateur utilise les règles suivantes dans un contexte d'annotation nullable désactivé :

- Vous ne pouvez pas déclarer de références nullables dans un contexte désactivé.
- Une valeur NULL peut être assignée à toutes les variables de référence.
- Aucun avertissement n'est généré lorsqu'une variable d'un type référence est déréférencée.
- L'opérateur null-forgiving ne peut pas être utilisé dans un contexte désactivé.

Le comportement est le même que dans les versions précédentes de C#.

Le compilateur utilise les règles suivantes dans un contexte d'annotation nullable activé :

- Toute variable d'un type référence est une **référence non nullable**.
- Toute référence non nullable peut être déréférencée sans problème.
- Tout type référence nullable (indiqué par `?` après le type dans la déclaration de variable) peut être null.

L'analyse statique détermine si la valeur est connue pour être non null lorsqu'elle est déréférencée. Si ce n'est pas le cas, le compilateur vous avertit.
- Vous pouvez utiliser l'opérateur null-forgiving pour déclarer qu'une référence nullable n'est pas null.

Dans un contexte d'annotation nullable activé, le caractère `?` ajouté à un type référence déclare un **type référence nullable**. L' **opérateur null-indulgent avec** `!` peut être ajouté à une expression pour déclarer que l'expression n'est pas null.

Contexte d'avertissement nullable

Le contexte d'avertissement nullable est différent du contexte d'annotation nullable. Des avertissements peuvent être activés, même lorsque les nouvelles annotations sont désactivées. Le compilateur utilise l'analyse de flux statique pour déterminer l'**état null** de toutes les références. L'état null est **non null** ou **peut-être null** quand le *contexte d'avertissement nullable* n'est pas **désactivé**. Si vous déréférencez une référence quand le compilateur a déterminé qu'elle est **peut-être null**, le compilateur vous avertit. L'état d'une référence est **peut-être null**, sauf si le compilateur peut déterminer une des deux conditions :

1. Une valeur non null a été assignée définitivement à la variable.
2. La variable ou l'expression a été vérifiée pour savoir si sa valeur est null avant de la référencer.

Le compilateur génère des avertissements lorsque vous déréférencez une variable ou une expression qui est **peut-être null** dans un contexte d'avertissement Nullable. En outre, le compilateur génère des avertissements quand une variable de type référence non null reçoit une variable ou une expression de type **null** dans un contexte d'annotation Nullable activé.

Les attributs décrivent des API

Vous ajoutez des attributs aux API qui fournissent au compilateur plus d'informations sur le moment où les arguments ou les valeurs de retour peuvent ou ne peuvent pas être null. Vous pouvez en savoir plus sur ces attributs dans notre article dans la référence du langage couvrant les attributs Nullable. Ces attributs sont ajoutés aux bibliothèques .NET sur les versions actuelles et à venir. Les API les plus couramment utilisées sont mises à jour en premier.

Pièges connus

Les tableaux et les structs qui contiennent des types référence sont des pièges connus dans la fonctionnalité de types référence Nullable.

Structures

Un struct qui contient des types référence n'acceptant pas les valeurs NULL permet de l'assigner `default` sans avertissement. Prenons l'exemple suivant :

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

Dans l'exemple précédent, il n'y a pas d'avertissement dans `PrintStudent(default)` tandis que les types de référence non Nullable `FirstName` et `LastName` sont null.

Un autre cas plus courant est lorsque vous traitez des structs génériques. Prenons l'exemple suivant :

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

Dans l'exemple précédent, la propriété `Bar` va être `null` au moment de l'exécution et elle est assignée à une chaîne non Nullable sans avertissement.

Tableaux

Les tableaux sont également un piège connu dans les types de référence Nullable. Prenons l'exemple suivant qui ne génère pas d'avertissements :

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

Dans l'exemple précédent, la déclaration du tableau indique qu'elle contient des chaînes qui n'acceptent pas les valeurs NULL, tandis que ses éléments sont tous initialisés à la valeur null. Ensuite, `s` une valeur null est assignée à la variable (le premier élément du tableau). Enfin, la variable `s` est déréférencée et provoque une exception Runtime.

Voir aussi

- Spécification des types de référence Nullable Draft
- Tutoriel d'introduction aux références nullables
- Migration d'une base de code existante vers des références nullables
- -Nullable (option du compilateur C#)

Mettre à jour les bibliothèques pour utiliser des types référence Nullable et communiquer des règles Nullable aux appelants

16/12/2020 • 22 minutes to read • Edit Online

L'ajout de types de référence Nullable signifie que vous pouvez déclarer si une `null` valeur est autorisée ou non pour chaque variable. En outre, vous pouvez appliquer un certain nombre d'attributs : `AllowNull` , `DisallowNull` ,,,, `MaybeNull` `NotNull` `NotNullWhen` `MaybeNullWhen` et `NotNullIfNotNull` pour décrire complètement les États null de l'argument et les valeurs de retour. Cela offre une excellente expérience lorsque vous écrivez du code. Vous recevez des avertissements si une variable qui n'accepte pas les valeurs NULL peut avoir la valeur `null` . Vous recevez des avertissements si une variable Nullable n'est pas vérifiée par null avant d'être déréférencée. La mise à jour de vos bibliothèques peut prendre du temps, mais les avantages le justifient. Plus d'informations que vous fournissez au compilateur sur les *cas où* une `null` valeur est autorisée ou interdite, les meilleurs avertissements que les utilisateurs de votre API obtiendront. Commençons par un exemple familier. Imaginez que votre bibliothèque possède l'API suivante pour récupérer une chaîne de ressource :

```
bool TryGetMessage(string key, out string message)
```

L'exemple précédent suit le `Try*` modèle familier dans .net. Il existe deux arguments de référence pour cette API : `key` et le `message` paramètre. Cette API comporte les règles suivantes relatives à la valeur null de ces arguments :

- Les appelants ne doivent pas passer `null` comme argument pour `key` .
- Les appelants peuvent passer une variable dont la valeur est `null` l'argument pour `message` .
- Si la `TryGetMessage` méthode retourne `true` , la valeur de `message` n'est pas null. Si la valeur de retour est `false,` la valeur de `message` (et son état null) est null.

La règle pour `key` peut être entièrement exprimée par le type de variable : `key` doit être un type de référence non Nullable. Le `message` paramètre est plus complexe. Elle autorise `null` comme argument, mais garantit que, en cas de réussite, cet `out` argument n'est pas null. Pour ces scénarios, vous avez besoin d'un vocabulaire plus riche pour décrire les attentes.

La mise à jour de votre bibliothèque pour les références Nullable nécessite plus que l'extinction `?` sur certaines variables et certains noms de types. L'exemple précédent montre que vous devez examiner vos API et prendre en compte vos attentes pour chaque argument d'entrée. Considérez les garanties pour la valeur de retour et `out` les `ref` arguments ou sur le retour de la méthode. Ensuite, communiquez ces règles au compilateur et le compilateur fournira des avertissements quand les appelants ne respectent pas ces règles.

Ce travail prend du temps. Commençons par des stratégies pour rendre votre bibliothèque ou votre application prenant en charge les valeurs NULL, tout en équilibrant d'autres exigences. Vous verrez comment équilibrer le développement en cours et activer les types de référence Nullable. Vous découvrirez les défis liés aux définitions de type générique. Vous apprendrez à appliquer des attributs pour décrire des conditions préalables et postérieures à des API individuelles.

Choisir une stratégie pour les types de référence Nullable

Le premier choix est si les types de référence Nullable doivent être activés ou désactivés par défaut. Vous avez deux stratégies :

- Activez les types de référence Nullable pour l'ensemble du projet et désactivez-le dans le code qui n'est pas prêt.
- Activez uniquement les types référence Nullable pour le code qui a été annoté pour les types référence Nullable.

La première stratégie fonctionne mieux lorsque vous ajoutez d'autres fonctionnalités à la bibliothèque lors de sa mise à jour pour les types de référence Nullable. Tout nouveau développement prend en charge les valeurs NULL. Lorsque vous mettez à jour le code existant, vous activez les types de référence Nullable dans ces classes.

Après cette première stratégie, procédez comme suit :

1. Activez les types de référence Nullable pour l'ensemble du projet en ajoutant l' `<Nullable>enable</Nullable>` élément à vos fichiers *csproj* .
2. Ajoutez le `#nullable disable` pragma à chaque fichier source de votre projet.
3. Lorsque vous travaillez sur chaque fichier, supprimez le pragma et résolvez tous les avertissements.

Cette première stratégie a plus de travail initial pour ajouter le pragma à chaque fichier. L'avantage est que chaque nouveau fichier de code ajouté au projet sera activé pour la valeur null. Tout nouveau travail prend en charge les valeurs null ; seul le code existant doit être mis à jour.

La deuxième stratégie fonctionne mieux si la bibliothèque est stable, et l'objectif principal du développement est d'adopter des types de référence Nullable. Vous activez les types de référence Nullable quand vous annotez des API. Lorsque vous avez terminé, vous activez les types de référence Nullable pour l'ensemble du projet.

Après cette seconde stratégie, procédez comme suit :

1. Ajoutez le `#nullable enable` pragma au fichier que vous souhaitez rendre compatible avec la valeur null.
2. Résolvez tous les avertissements.
3. Poursuivez ces deux premières étapes jusqu'à ce que vous ayez pris en charge la valeur null de la bibliothèque entière.
4. Activez les types Nullable pour l'ensemble du projet en ajoutant l' `<Nullable>enable</Nullable>` élément à vos fichiers *csproj* .
5. Supprimez les `#nullable enable` pragmas, car ils ne sont plus nécessaires.

Cette deuxième stratégie a moins de travail avant. Le compromis est que la première tâche lorsque vous créez un nouveau fichier consiste à ajouter le pragma et à le rendre compatible avec la valeur null. Si des développeurs de votre équipe oublient, ce nouveau code est désormais dans le backlog de travail pour rendre tout le code prenant en charge la valeur null.

Les stratégies que vous choisissez dépendent de la quantité de développement actif dans votre projet. Plus votre projet est mature et stable, plus la seconde stratégie est performante. Plus vous développez de fonctionnalités, mieux la première stratégie.

IMPORTANT

Le contexte Nullable global ne s'applique pas aux fichiers de code générés. Dans l'une ou l'autre stratégie, le contexte Nullable est *désactivé* pour tout fichier source marqué comme généré. Cela signifie que toutes les API dans les fichiers générés ne sont pas annotées. Un fichier est marqué comme généré de quatre façons :

1. Dans le fichier. editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
2. Placez `<auto-generated>` ou `<auto-generated/>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.
3. Démarrer le nom de fichier avec *TemporaryGeneratedFile_*
4. Terminez le nom de fichier avec *. Designer.cs*, *. generated.cs*, *. g.cs* ou *. g.i.cs*.

Les générateurs peuvent s'abonner à l'aide de la `#nullable` directive de préprocesseur.

Les avertissements de type Nullable introduisent-ils des modifications avec rupture ?

Avant d'activer les types de référence Nullable, les variables sont considérées comme *Nullable oublie*. Une fois que vous activez les types de référence Nullable, toutes ces variables ne peuvent pas avoir la valeur *null*. Le compilateur émet des avertissements si ces variables ne sont pas initialisées à des valeurs non null.

Une autre source probable d'avertissements est celle des valeurs de retour lorsque la valeur n'a pas été initialisée.

La première étape de la résolution des avertissements du compilateur consiste à utiliser `?` des annotations sur les types de paramètres et de retour pour indiquer quand les arguments ou les valeurs de retour peuvent être null. Lorsque les variables de référence ne doivent pas avoir la valeur null, la déclaration d'origine est correcte. Au fur et à mesure que vous effectuez cette tâche, votre objectif n'est pas simplement de corriger les avertissements. L'objectif le plus important est de faire en sorte que le compilateur comprenne votre intention pour les valeurs NULL potentielles. Lorsque vous examinez les avertissements, vous atteignez votre prochaine décision majeure pour votre bibliothèque. Voulez-vous envisager de modifier les signatures d'API pour communiquer plus clairement votre intention de conception ? Une meilleure signature d'API pour la `TryGetMessage` méthode examinée précédemment peut être :

```
string? TryGetMessage(string key);
```

La valeur de retour indique la réussite ou l'échec et transporte la valeur si la valeur a été trouvée. Dans de nombreux cas, la modification des signatures d'API peut améliorer la façon dont elles communiquent les valeurs NULL.

Toutefois, pour les bibliothèques publiques ou les bibliothèques avec des bases d'utilisateurs volumineuses, vous préférerez peut-être ne pas introduire de modifications de signature d'API. Dans ces cas-là, et d'autres modèles courants, vous pouvez appliquer des attributs pour définir plus clairement quand un argument ou une valeur de retour peut être `null` . Si vous envisagez de modifier la surface de votre API, vous constaterez sans doute que les annotations de type ne suffisent pas pour décrire les `null` valeurs des arguments ou des valeurs de retour. Dans ces instances, vous pouvez appliquer des attributs pour décrire plus clairement une API.

Les attributs étendent les annotations de type

Plusieurs attributs ont été ajoutés pour exprimer des informations supplémentaires sur l'État null des variables. Tout le code que vous avez écrit avant C# 8 a introduit des types de référence Nullable était *null oublie*. Cela signifie que toute variable de type référence peut avoir la valeur null, mais que les vérifications de valeur NULL ne sont pas requises. Une fois que votre code prend en *charge les valeurs NULL*, ces règles changent. Les types référence ne doivent jamais être la `null` valeur, et les types référence Nullable doivent être vérifiés `null` avant

d'être déréférencés.

Les règles de vos API sont probablement plus compliquées, comme vous l'avez vu dans le `TryGetValue` scénario d'API. La plupart de vos API ont des règles plus complexes pour le moment où les variables peuvent ou ne peuvent pas être `null` . Dans ce cas, vous utiliserez des attributs pour exprimer ces règles. Les attributs qui décrivent la sémantique de votre API se trouvent dans l'article sur les attributs qui affectent l'analyse Nullable.

Définitions génériques et possibilité de valeur null

La communication correcte de l'État null des types génériques et des méthodes génériques requiert une attention particulière. Les soins supplémentaires proviennent du fait qu'un type valeur Nullable et un type référence Nullable sont fondamentalement différents. Un `int?` est un synonyme de `Nullable<int>` , tandis que `string?` est `string` associé à un attribut ajouté par le compilateur. Le résultat est que le compilateur ne peut pas générer de code correct pour `T?` sans savoir si `T` est un `class` ou un `struct` .

Ce fait ne signifie pas que vous ne pouvez pas utiliser un type Nullable (type valeur ou type référence) comme argument de type pour un type générique fermé. `List<string?>` Et `List<int?>` sont des instanciations valides de `List<T>` .

Cela signifie que vous ne pouvez pas utiliser `T?` dans une déclaration de classe ou de méthode générique sans contraintes. Par exemple, Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>) ne peut pas être modifié pour retourner `T?` . Vous pouvez contourner cette limitation en ajoutant la `struct` `class` contrainte ou. Avec l'une de ces contraintes, le compilateur sait comment générer du code pour `T` et `T?` .

Vous pouvez restreindre les types utilisés pour un argument de type générique à des types non nullables. Pour ce faire, vous pouvez ajouter la `notnull` contrainte sur cet argument de type. Quand cette contrainte est appliquée, l'argument de type ne doit pas être un type Nullable.

Propriétés initialisées tardivement, objets Transfert de données et possibilité de valeur null

Indiquant la possibilité de valeur null des propriétés qui sont initialisées tardivement, ce qui signifie qu'elles sont définies après la construction, peut nécessiter une attention particulière afin de garantir que votre classe continue d'exprimer correctement l'intention de conception d'origine.

Les types qui contiennent des propriétés initialisées tardivement, telles que les objets Transfert de données (DTO), sont souvent instanciés par une bibliothèque externe, comme un ORM de base de données (mappage relationnel d'objets), un désérialiseur ou un autre composant qui remplit automatiquement les propriétés d'une autre source.

Considérez la classe DTO suivante, avant d'activer les types de référence Nullable, qui représente un Student :

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string VehicleRegistration { get; set; }
}
```

L'intention de conception (indiquée dans ce cas par l' `Required` attribut) suggère que dans ce système, `FirstName` les `LastName` Propriétés et sont **obligatoires** et, par conséquent, non null.

La `VehicleRegistration` propriété n'étant **pas obligatoire**, elle peut avoir la valeur null.

Lorsque vous activez les types de référence Nullable, vous souhaitez indiquer quelles propriétés de votre DTO peuvent être Nullable, conformément à votre intention d'origine :

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

Pour ce DTO, la seule propriété qui peut être null est `VehicleRegistration` .

Toutefois, le compilateur déclenche `CS8618` des avertissements pour `FirstName` et `LastName` , indiquant que les propriétés non Nullable ne sont pas initialisées.

Trois options sont à votre disposition pour résoudre les avertissements du compilateur de manière à conserver l'intention d'origine. L'une de ces options est valide. vous devez choisir celui qui convient le mieux à vos exigences de style et de conception de codage.

Initialiser dans le constructeur

Pour résoudre les avertissements non initialisés, la méthode idéale consiste à initialiser les propriétés dans le constructeur :

```
class Student
{
    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

Cette approche fonctionne uniquement si la bibliothèque que vous utilisez pour instancier la classe prend en charge le passage de paramètres dans le constructeur.

Une bibliothèque peut prendre en charge la transmission de *certaines* propriétés dans le constructeur, mais pas toutes. Par exemple, EF Core prend en charge la liaison de constructeur pour les propriétés de colonne normales, mais pas pour les propriétés de navigation.

Consultez la documentation sur la bibliothèque qui instancie votre classe pour comprendre la portée à laquelle elle prend en charge la liaison de constructeur.

Propriété avec champ de stockage Nullable

Si la liaison de constructeur ne fonctionne pas pour vous, une façon de traiter ce problème consiste à avoir une propriété qui n'accepte pas les valeurs NULL avec un champ de stockage Nullable :

```
private string? _firstName;

[Required]
public string FirstName
{
    set => _firstName = value;
    get => _firstName
           ?? throw new InvalidOperationException("Uninitialized " + nameof(FirstName))
}
```

Dans ce scénario, si `FirstName` vous accédez à la propriété avant qu'elle n'ait été initialisée, le code lève une `InvalidOperationException` , car le contrat d'API a été utilisé de façon incorrecte.

Sachez que certaines bibliothèques peuvent avoir des considérations particulières lors de l'utilisation de champs de stockage. Par exemple, EF Core devrez peut-être être configuré pour utiliser correctement les champs de sauvegarde .

Initialise la propriété avec la valeur null

Comme une alternative terser à l'utilisation d'un champ de stockage Nullable, ou si la bibliothèque qui instancie votre classe n'est pas compatible avec cette approche, vous pouvez initialiser la propriété `null` directement, à l'aide de l'opérateur null-indulgent avec (`!`) :

```
[Required]
public string FirstName { get; set; } = null!;

[Required]
public string LastName { get; set; } = null!;

public string? VehicleRegistration { get; set; }
```

Vous n'observerez jamais une valeur null réelle au moment de l'exécution sauf en raison d'un bogue de programmation, en accédant à la propriété avant qu'elle ait été correctement initialisée.

Voir aussi

- Migration d'une base de code existante vers des références nullables
- Utilisation des types de référence Nullable dans EF Core

Espaces de noms (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Les espaces de noms sont largement utilisés de deux façons dans la programmation avec C#. Tout d'abord, .NET utilise des espaces de noms pour organiser ses nombreuses classes, comme suit :

```
System.Console.WriteLine("Hello World!");
```

System est un espace de noms et Console est une classe de cet espace de noms. Vous pouvez utiliser le mot clé `using`. Ainsi, le nom complet n'est pas obligatoire, comme dans l'exemple suivant :

```
using System;
```

```
Console.WriteLine("Hello World!");
```

Pour plus d'informations, consultez Directive using.

Deuxièmement, la déclaration de vos propres espaces de noms peut vous aider à contrôler l'étendue des noms de classes et de méthodes dans les projets de programmation plus vastes. Utilisez le mot clé namespace pour déclarer un espace de noms, comme dans l'exemple suivant :

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

Le nom de l'espace de noms doit être un nom d'identificateur C# valide.

Vue d'ensemble des espaces de noms

Les espaces de noms ont les propriétés suivantes :

- Ils organisent les projets de code de taille importante.
- Ils sont délimités à l'aide de l'opérateur `.`.
- La directive `using` évite de devoir spécifier le nom de l'espace de noms pour chaque classe.
- L'espace de noms `global` est l'espace de noms « racine » : `global::System` fait toujours référence à l'espace de noms System .NET.

spécification du langage C#

Pour plus d'informations, voir la section Espace de noms de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Utilisation d'espaces de noms
- Comment utiliser l'espace de noms My
- Noms d'identificateurs
- using, directive
- ::, Opérateur

Types, variables et valeurs

16/12/2020 • 12 minutes to read • Edit Online

C# est un langage fortement typé. Chaque variable et chaque constante ont un type, tout comme chaque expression qui fournit une valeur. Chaque signature de méthode spécifie un type pour chaque paramètre d'entrée et pour la valeur de retour. La bibliothèque de classes .NET définit un ensemble de types numériques intégrés, ainsi que des types plus complexes qui représentent un large éventail de constructions logiques, telles que le système de fichiers, les connexions réseau, les collections et tableaux d'objets, et les dates. Un programme C# classique utilise des types fournis dans la bibliothèque de classes, mais aussi des types définis par l'utilisateur qui modélisent les concepts propres au domaine du problème du programme.

Un type peut stocker les informations suivantes :

- Espace de stockage nécessaire pour une variable du type.
- Valeurs minimale et maximale que le type peut représenter.
- Membres (méthodes, champs, événements, etc.) que le type contient.
- Type de base dont le type est hérité.
- Interface (s) qu'il implémente.
- Emplacement où sera allouée la mémoire pour les variables au moment de l'exécution.
- Sortes d'opérations autorisées.

Le compilateur utilise les informations de type pour s'assurer que toutes les opérations qui sont effectuées dans votre code sont de *type safe*. Par exemple, si vous déclarez une variable de type int, le compilateur vous permet d'utiliser la variable dans des opérations d'addition et de soustraction. Si vous essayez d'effectuer ces mêmes opérations avec une variable de type bool, le compilateur génère une erreur, comme illustré dans l'exemple suivant :

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> Développeurs C et C++ : notez que dans C#, bool n'est pas convertible en int.

Le compilateur incorpore les informations de type dans le fichier exécutable sous forme de métadonnées. Le common language runtime (CLR) utilise ces métadonnées au moment de l'exécution pour garantir que le type est sécurisé lorsqu'il alloue et libère de la mémoire.

Spécification de types dans les déclarations de variable

Lorsque vous déclarez une variable ou une constante dans un programme, vous devez spécifier son type ou utiliser le mot clé var pour permettre au compilateur de déduire le type. L'exemple suivant montre des déclarations de variable qui utilisent des types numériques intégrés et des types complexes définis par l'utilisateur :

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Les types de paramètres de méthode et de valeurs de retour sont spécifiés dans la signature de méthode. La signature suivante présente une méthode qui requiert int comme argument d'entrée et retourne une chaîne :

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Après qu'une variable est déclarée, elle ne peut pas être déclarée de nouveau avec un nouveau type et elle ne peut pas être définie sur une valeur qui n'est pas compatible avec son type déclaré. Par exemple, vous ne pouvez pas déclarer un int et lui assigner une valeur booléenne `true` . Toutefois, les valeurs peuvent être converties en d'autres types, par exemple, quand elles sont assignées à de nouvelles variables ou passées comme arguments de méthode. Une *conversion de type* qui ne cause pas de perte de données est effectuée automatiquement par le compilateur. Une conversion susceptible de causer la perte de données exige une variable *cast* dans le code source.

Pour plus d'informations, consultez Cast et conversions de types.

Types intégrés

C# fournit un jeu standard de types numériques intégrés pour représenter les nombres entiers, les valeurs à virgule flottante, les expressions booléennes, les caractères de texte, les valeurs décimales et d'autres types de données. Il existe également des types **string** et **object** intégrés, que vous pouvez utiliser dans n'importe quel programme C#. Pour obtenir la liste complète des types intégrés, consultez types intégrés.

Types personnalisés

Vous utilisez les constructions struct, classe, interface et enum pour créer vos propres types personnalisés. La bibliothèque de classes .NET est une collection de types personnalisés fournie par Microsoft que vous pouvez utiliser dans vos propres applications. Par défaut, les types les plus fréquemment utilisés dans la bibliothèque de classes sont disponibles dans tous les programmes C#. Les autres types sont disponibles uniquement si vous ajoutez explicitement une référence de projet à l'assembly dans lequel ils sont définis. Une fois que le compilateur a une référence à l'assembly, vous pouvez déclarer des variables (et des constantes) des types déclarés dans cet assembly dans le code source.

Types génériques

Un type peut être déclaré avec un ou plusieurs *paramètres de type* qui servent d'espace réservé pour le type réel (le *type concret*) que le code client fournit lorsqu'il crée une instance du type. Ces types sont appelés *types génériques*. Par exemple, List<T> a un paramètre de type qui, par Convention, reçoit le nom *T*. Lorsque vous créez

une instance du type, vous spécifiez le type des objets que la liste doit contenir, par exemple String :

```
List<string> strings = new List<string>();
```

L'utilisation du paramètre de type rend possible la réutilisation de la même classe pour contenir tout type d'élément, sans avoir à convertir chaque élément en object. Les classes de collection génériques sont appelées *collections fortement typées* , car le compilateur connaît le type spécifique des éléments de la collection et peut déclencher une erreur au moment de la compilation si, par exemple, vous tentez d'ajouter un entier à l' `strings` objet dans l'exemple précédent. Pour plus d'informations, consultez Génériques.

Types implicites, types anonymes et types tuple

Comme indiqué précédemment, vous pouvez attribuer implicitement un type à une variable locale (mais pas les membres de la classe) à l'aide du mot clé var. La variable reçoit toujours un type au moment de la compilation, mais le type est fourni par le compilateur. Pour plus d'informations, consultez variables locales implicitement typées.

Dans certains cas, il est difficile de créer un type nommé pour des ensembles simples de valeurs associées que vous ne souhaitez pas stocker ou transférer en dehors des limites de la méthode. Vous pouvez alors créer des *types anonymes*. Pour plus d'informations, consultez types anonymes.

Il est courant de vouloir retourner plusieurs valeurs d'une méthode. Vous pouvez créer des *types tuple* qui retournent plusieurs valeurs dans un seul appel de méthode. Pour plus d'informations, consultez types de tuples.

Système de type commun (CTS)

Il est important de comprendre deux points fondamentaux à propos du système de type dans le .NET :

- Il prend en charge le principe d'héritage. Les types peuvent dériver d'autres types, appelés *types de base*. Le type dérivé hérite (avec certaines restrictions) des méthodes, des propriétés et des autres membres du type de base. Le type de base peut à son tour dériver d'un autre type, auquel cas le type dérivé hérite des membres des deux types de base dans sa hiérarchie d'héritage. Tous les types, y compris les types numériques intégrés tels que Int32 (mot clé C# : `int`), dérivent au final d'un seul type de base, qui est Object (mot clé C# : `object`). Cette hiérarchie de types unifiée est appelée système de type commun (CTS, Common type System). Pour plus d'informations sur l'héritage dans C#, consultez Héritage.
- Chaque type du CTS est défini comme *type valeur* ou *type référence*. Cela inclut tous les types personnalisés dans la bibliothèque de classes .NET, ainsi que les types définis par l'utilisateur. Les types que vous définissez à l'aide du `struct` `enum` mot clé ou sont des types valeur. Pour plus d'informations sur les types valeur, consultez types valeur. Les types que vous définissez à l'aide du mot clé class sont des types référence. Pour plus d'informations sur les types référence, consultez Classes. Les types référence et les types valeur ne suivent pas les mêmes règles de compilation et ont un comportement différent au moment de l'exécution.

Voir aussi

- Types de structure
- Types d'énumération
- Classes

Classes (Guide de programmation C#)

16/12/2020 • 9 minutes to read • Edit Online

Types référence

Un type défini comme class est un *type référence*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur Null tant que vous n'avez pas explicitement créé une instance de la classe à l'aide de l'opérateur new, ou que vous ne lui avez pas assigné un objet existant d'un type compatible, comme indiqué dans l'exemple suivant :

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

Quand l'objet est créé, une quantité de mémoire suffisante est allouée sur le tas managé de l'objet spécifié, et la variable contient uniquement une référence à l'emplacement de cet objet. Les types sur le tas managé entraînent une surcharge quand ils sont alloués et récupérés par la fonctionnalité de gestion automatique de la mémoire du CLR (appelée *garbage collection*). Toutefois, le garbage collection est également optimisé et, dans la plupart des cas, ne nuit pas aux performances. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire et garbage collection.

Déclaration de classes

Les classes sont déclarées à l'aide du mot clé class suivi d'un identificateur unique, comme l'illustre l'exemple suivant :

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

Le mot clé `class` est précédé du niveau d'accès. Comme public est utilisé dans ce cas, n'importe qui peut créer des instances de cette classe. Le nom de la classe suit le mot clé `class`. Le nom de la classe doit être un nom d'identificateur C# valide. Le reste de la définition est le corps de la classe, où le comportement et les données sont définis. Les champs, propriétés, méthodes et événements d'une classe sont désignés collectivement par le terme « *membres de classe* ».

Création d'objets

Bien qu'ils soient parfois employés indifféremment, une classe et un objet sont deux choses différentes. Une classe définit un type d'objet, mais il ne s'agit pas d'un objet en soi. Un objet, qui est une entité concrète basée sur une classe, est parfois désigné par le terme « instance de classe ».

Vous pouvez créer des objets en utilisant le mot clé new suivi du nom de la classe sur laquelle l'objet est basé, comme suit :

```
Customer object1 = new Customer();
```

Quand une instance d'une classe est créée, une référence à l'objet est repassée au programmeur. Dans l'exemple précédent, `object1` est une référence à un objet basé sur `Customer`. Cette référence fait référence au nouvel objet, mais elle ne contient pas ses données. En fait, vous pouvez créer une référence d'objet sans créer d'objet :

```
Customer object2;
```

Nous vous déconseillons de créer des références d'objet comme celle-ci, sans référence à un objet, car toute tentative d'accès à un objet à l'aide d'une telle référence échoue au moment de l'exécution. Toutefois, une telle référence peut être faite pour faire référence à un objet, soit en créant un nouvel objet, soit en lui assignant un objet existant, tel que celui-ci :

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Ce code crée deux références d'objet qui font toutes deux référence au même objet. Toute modification apportée à l'objet par le biais de `object3` est donc reflétée dans les utilisations suivantes de `object4`. Les objets qui sont basés sur des classes étant désignés par référence, les classes sont appelées des « types référence ».

Héritage de classe

Les classes prennent entièrement en charge l' *héritage* , caractéristique fondamentale de la programmation orientée objet. Lorsque vous créez une classe, vous pouvez hériter de toute autre classe qui n'est pas définie comme sealed, et d'autres classes peuvent hériter de votre classe et remplacer les méthodes virtuelles de la classe. En outre, vous pouvez implémenter une ou plusieurs interfaces.

L'héritage se fait par le biais d'une *dérivation* , ce qui signifie qu'une classe est déclarée à l'aide d'une *classe de base* dont elle hérite les données et le comportement. Pour spécifier une classe de base, ajoutez deux-points et le nom de la classe de base après le nom de la classe dérivée, comme suit :

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Quand une classe déclare une classe de base, elle hérite de tous les membres de la classe de base à l'exception des constructeurs. Pour plus d'informations, consultez Héritage.

Contrairement à C++, une classe en C# ne peut hériter directement que d'une classe de base. Toutefois, une classe de base pouvant elle-même hériter d'une autre classe, une classe peut hériter indirectement de plusieurs classes de base. En outre, une classe peut implémenter directement une ou plusieurs interfaces. Pour plus d'informations, consultez Interfaces.

Une classe peut être déclarée abstract. Une classe abstraite contient des méthodes abstraites qui ont une définition de signature, mais aucune implémentation. Les classes abstraites ne peuvent pas être instanciées. Elles peuvent être utilisées uniquement à travers des classes dérivées qui implémentent les méthodes abstraites. En revanche, une classe sealed ne permet pas à d'autres classes de dériver d'elle. Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Les définitions de classe peuvent être fractionnées entre différents fichiers sources. Pour plus d'informations, consultez la page Classes et méthodes partielles.

Exemple

L'exemple suivant définit une classe publique qui contient une propriété implémentée automatiquement, une méthode et une méthode spéciale appelée un constructeur. Pour plus d'informations, consultez les rubriques Propriétés, Méthodes et Constructeurs. Les instances de la classe sont ensuite instanciées à l'aide du mot clé `new`.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#

- Programmation orientée objet
- Polymorphisme
- Noms d’identificateurs
- Members (Membres)
- Méthodes
- Constructeurs
- Finaliseurs
- Objets

Déconstruction de tuples et d'autres types

16/12/2020 • 17 minutes to read • Edit Online

Un tuple offre un moyen léger de récupérer plusieurs valeurs à partir d'un appel de méthode. Cependant, une fois que vous récupérez le tuple, vous devez gérer ses éléments individuels. Cette gestion élément par élément est assez fastidieuse, comme le montre l'exemple suivant. La méthode `QueryCityData` retourne un tuple de 3 éléments, et chacun de ses éléments est affecté à une variable dans une opération distincte.

```
using System;

public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

La récupération de plusieurs valeurs de champs et de propriétés d'un objet peut également être assez fastidieuse : vous devez affecter une valeur de champ ou de propriété à une variable membre par membre.

À compter de C# 7.0, vous pouvez récupérer plusieurs éléments d'un tuple ou récupérer plusieurs valeurs de champ et de propriété, ainsi que des valeurs calculées, à partir d'un objet en une seule opération de *déconstruction*. Quand vous déconstruisez un tuple, vous affectez ses éléments à des variables individuelles. Quand vous déconstruisez un objet, vous affectez des valeurs sélectionnées à des variables individuelles.

Déconstruction d'un tuple

Des fonctionnalités C# intégrées prennent en charge la déconstruction des tuples, ce qui vous permet de décomposer tous les éléments d'un tuple en une seule opération. La syntaxe générale de déconstruction d'un tuple est similaire à la syntaxe qui permet d'en définir un : vous placez les variables auxquelles chaque élément doit être affecté entre des parenthèses, dans la partie gauche d'une instruction d'affectation. Par exemple, l'instruction suivante affecte les éléments d'un tuple de 4 éléments à quatre variables distinctes :

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

Il existe trois façons de déconstruire un tuple :

- Vous pouvez déclarer explicitement le type de chaque champ entre des parenthèses. L'exemple suivant utilise cette approche pour déconstruire un tuple de 3 éléments retourné par la méthode `QueryCityData`.

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- Vous pouvez utiliser le mot clé `var` pour que C# infère le type de chaque variable. Vous placez le mot clé `var` en dehors des parenthèses. L'exemple suivant utilise l'inférence de type lors de la déconstruction du tuple de 3 éléments retourné par la méthode `QueryCityData`.

  ```
  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  Vous pouvez aussi utiliser le mot clé `var` individuellement avec tout ou partie des déclarations de variables à l'intérieur des parenthèses.

  ```
  public static void Main()
  {
      (string city, var population, var area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  Ceci est assez fastidieux et n'est pas recommandé.

- Enfin, vous pouvez déconstruire le tuple dans des variables qui ont déjà été déclarées.

  ```
  public static void Main()
  {
      string city = "Raleigh";
      int population = 458880;
      double area = 144.8;

      (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

Notez que vous ne pouvez pas spécifier un type spécifique en dehors des parenthèses, même si tous les champs du tuple ont le même type. Cela génère l'erreur du compilateur CS8136, « La déconstruction de 'var (...)' form interdit un type spécifique pour 'var'.».

Notez que vous devez également affecter chaque élément du tuple à une variable. Si vous omettez des éléments, le compilateur génère l'erreur CS8132, « Impossible de déconstruire un tuple de « x » éléments en « y » variables ».

Notez que vous ne pouvez pas mélanger des déclarations et des affectations à des variables existantes sur le côté gauche d'une déconstruction. Le compilateur génère l'erreur CS8184, « Une déconstruction ne peut pas contenir à la fois des déclarations et des expressions à gauche », quand les membres incluent des variables nouvellement déclarées et des variables existantes.

Déconstruction d'éléments d'un tuple en ignorant des éléments

Souvent, lors de la déconstruction d'un tuple, vous êtes intéressé seulement par les valeurs de certains éléments. À compter de C# 7.0, vous pouvez tirer parti de la prise en charge des *éléments ignorés* , qui sont des variables en écriture seule dont vous avez choisi d'ignorer les valeurs. Un élément ignoré est désigné par un caractère de soulignement (« _ ») dans une affectation. Vous pouvez ignorer autant de valeurs que vous le souhaitez ; pour représenter toutes les valeurs, utilisez l'élément ignoré unique, `_`.

L'exemple suivant illustre l'utilisation de tuples avec des éléments ignorés. La méthode `QueryCityDataForYears` retourne un tuple de 6 éléments avec le nom d'une ville, sa région, une année, la population de la ville pour cette année, une seconde année et la population de la ville pour cette seconde année. L'exemple montre la différence de population entre ces deux années. Parmi les données disponibles dans le tuple, nous ne sommes pas intéressés par la région de la ville, et nous connaissons le nom de la ville et les deux dates au moment du design. Par conséquent, nous sommes intéressés seulement par les deux valeurs de la population stockées dans le tuple et nous pouvons gérer ses valeurs restantes comme éléments ignorés.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Déconstruction de types définis par l'utilisateur

C# n'offre pas de prise en charge intégrée pour déconstruire les types non-tuple. Cependant, en tant que créateur d'une classe, d'un struct ou d'une interface, vous pouvez permettre la déconstruction du type en implémentant une ou plusieurs méthodes `Deconstruct`. La méthode retourne void, et chaque valeur à déconstruire est indiquée par un paramètre out dans la signature de la méthode. Par exemple, la méthode `Deconstruct` suivante d'une classe `Person` retourne le prénom, le deuxième prénom et le nom :

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

Vous pouvez alors déconstruire une instance de la classe `Person` nommée `p` avec une affectation comme celle-ci :

```
var (fName, mName, lName) = p;
```

L'exemple suivant surcharge la méthode `Deconstruct` de façon retourner différentes combinaisons des propriétés d'un objet `Person`. Les différentes surcharges retournent :

- Un prénom et un nom.
- Prénom, deuxième prénom et nom de famille.
- Un prénom, un nom, un nom de ville et un nom d'état.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Comme vous pouvez surcharger la méthode `Deconstruct` de façon à refléter les groupes de données qui sont souvent extraits d'un objet, vous devez être attentif à définir des méthodes `Deconstruct` avec des signatures qui les distinguent et ne sont pas ambiguës. Plusieurs méthodes `Deconstruct` ayant même nombre de paramètres `out`, ou le même nombre et le même type de paramètres `out` dans un ordre différent, peuvent provoquer de la confusion.

La méthode `Deconstruct` surchargée dans l'exemple suivant montre une source possible de confusion. La

première surcharge retourne le prénom, le deuxième prénom, le nom et l'âge d'un objet `Person`, dans cet ordre. La seconde surcharge retourne les informations de nom seulement avec le revenu annuel, mais le prénom, le deuxième prénom et le nom sont dans un ordre différent. Ceci facilite la confusion dans l'ordre des arguments lors de la déconstruction d'une instance de `Person`.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public DateTime DateOfBirth { get; set; }
    public Decimal AnnualIncome { get; set; }

    public void Deconstruct(out string fname, out string mname, out string lname, out int age)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;

        // calculate the person's age
        var today = DateTime.Today;
        age = today.Year - DateOfBirth.Year;
        if (DateOfBirth.Date > today.AddYears(-age))
            age--;
    }

    public void Deconstruct(out string lname, out string fname, out string mname, out decimal income)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
        income = AnnualIncome;
    }
}
```

Déconstruction d'un type défini par l'utilisateur avec des éléments ignorés

Tout comme vous le faites avec des tuples, vous pouvez utiliser des éléments ignorés pour ignorer des éléments sélectionnés retournés par une méthode `Deconstruct`. Chaque élément ignoré est défini par une variable nommée « _ », et une même opération de déconstruction peut inclure plusieurs éléments ignorés.

L'exemple suivant déconstruit un objet `Person` en quatre chaînes (le prénom et le nom, la ville et l'état), mais ignore le nom et l'état.

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Déconstruction d'un type défini par l'utilisateur avec une méthode d'extension

Si vous n'avez pas créé une classe, un struct ou une interface, vous pouvez néanmoins déconstruire des objets de

ce type en implémentant une ou plusieurs `Deconstruct` méthodes d'extension pour retourner les valeurs qui vous intéressent.

L'exemple suivant définit deux méthodes d'extension `Deconstruct` pour la classe System.Reflection.PropertyInfo. La première retourne un ensemble de valeurs qui indique les caractéristiques de la propriété, notamment son type, si elle est statique ou d'instance, si elle est en lecture seule et si elle est indexée. La seconde indique l'accessibilité de la propriété. Comme l'accessibilité des accesseurs get et set peut varier, des valeurs booléennes indiquent si la propriété a des accesseurs get et set distincts et, le cas échéant, s'ils ont la même accessibilité. S'il n'existe qu'un seul accesseur, ou si les deux accesseurs get et set ont la même accessibilité, la variable `access` indique l'accessibilité de la propriété comme un tout. Sinon, l'accessibilité des accesseurs get et set est indiquée par les variables `getAccess` et `setAccess`.

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod;

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
```

```
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class Example
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name} property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
```

```
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item property: public
```

Voir aussi

- Éléments ignorés
- Types de tuples

Interfaces (Guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

Une interface contient des définitions pour un groupe de fonctionnalités connexes qu'une classe non abstraite ou un struct doit implémenter. Une interface peut définir des `static` méthodes, qui doivent avoir une implémentation. À compter de C# 8,0, une interface peut définir une implémentation par défaut pour les membres. Une interface ne peut pas déclarer des données d'instance, telles que des champs, des propriétés implémentées automatiquement ou des événements de type propriété.

En utilisant des interfaces, vous pouvez, par exemple, inclure le comportement de plusieurs sources dans une classe. Cette fonctionnalité est importante en C#, car le langage ne prend pas en charge l'héritage multiple de classes. De plus, vous devez utiliser une interface si vous voulez simuler l'héritage pour des structs, car ils ne peuvent en fait pas hériter d'un autre struct ou d'une autre classe.

Vous définissez une interface à l'aide du mot clé interface , comme le montre l'exemple suivant.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Le nom d'une interface doit être un nom d' identificateurC# valide. Par convention, les noms d'interface commencent par un `I` majuscule.

Toute classe ou tout struct qui implémentent l'interface IEquatable<T> doivent contenir une définition pour une méthode Equals qui correspond à la signature spécifiée par l'interface. Ainsi, vous pouvez compter sur une classe qui implémente `IEquatable<T>` pour contenir une méthode `Equals` avec laquelle une instance de la classe peut déterminer si elle est égale à une autre instance de la même classe.

La définition de `IEquatable<T>` ne fournit pas d'implémentation pour `Equals` . Une classe ou un struct peut implémenter plusieurs interfaces, mais une classe peut uniquement hériter d'une classe unique.

Pour plus d'informations sur les classes abstraites, consultez Classes abstract et sealed et membres de classe.

Les interfaces peuvent contenir des méthodes d'instance, des propriétés, des événements, des indexeurs ou toute combinaison de ces quatre types de membres. Les interfaces peuvent contenir des constructeurs statiques, des champs, des constantes ou des opérateurs. Pour obtenir des liens vers des exemples, consultez les Sections connexes. Une interface ne peut pas contenir de champs d'instance, de constructeurs d'instance ou de finaliseurs. Les membres d'interface sont publics par défaut.

Pour implémenter un membre d'interface, le membre correspondant de la classe d'implémentation doit être public, non statique et porter le même nom et la même signature que le membre d'interface.

Quand une classe ou un struct implémente une interface, la classe ou le struct doivent fournir une implémentation pour tous les membres que l'interface déclare, mais ne fournit pas d'implémentation par défaut pour. Toutefois, si une classe de base implémente une interface, toute classe dérivée de la classe de base hérite de cette implémentation.

L'exemple suivant illustre une implémentation de l'interface IEquatable<T>. La classe d'implémentation, `Car`, doit fournir une implémentation de la méthode Equals.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Les propriétés et indexeurs d'une classe peuvent définir des accesseurs supplémentaires pour une propriété ou un indexeur qui est défini dans une interface. Par exemple, une interface peut déclarer une propriété qui a un accesseur get. La classe qui implémente l'interface peut déclarer la même propriété avec à la fois un accesseur `get` et un accesseur set. Toutefois, si la propriété ou l'indexeur utilisent une implémentation explicite, les accesseurs doivent correspondre. Pour plus d'informations sur l'implémentation explicite, consultez les pages Implémentation d'interface explicite et Propriétés d'interface.

Les interfaces peuvent hériter d'une ou de plusieurs interfaces. L'interface dérivée hérite des membres de ses interfaces de base. Une classe qui implémente une interface dérivée doit implémenter tous les membres de l'interface dérivée, y compris tous les membres des interfaces de base de l'interface dérivée. Cette classe peut être implicitement convertie en une interface dérivée ou l'une de ses interfaces de base. Une classe peut inclure une interface plusieurs fois par le biais de classes de base qu'elle hérite ou d'interfaces dont d'autres interfaces héritent. Toutefois, la classe peut fournir une implémentation d'une interface une seule fois et uniquement si la classe déclare l'interface dans le cadre de la définition de la classe (`class ClassName : InterfaceName`). Si l'interface est héritée car vous avez hérité d'une classe de base qui implémente l'interface, la classe de base fournit l'implémentation des membres de l'interface. Toutefois, la classe dérivée peut réimplémenter des membres d'interface virtuels au lieu d'utiliser l'implémentation héritée. Quand les interfaces déclarent une implémentation par défaut d'une méthode, toute classe qui implémente cette interface hérite de cette implémentation. Les implémentations définies dans les interfaces sont virtuelles et la classe d'implémentation peut remplacer cette implémentation.

Une classe de base peut également implémenter des membres d'interface à l'aide de membres virtuels. Dans ce cas, une classe dérivée peut modifier le comportement de l'interface en substituant les membres virtuels. Pour plus d'informations sur les membres virtuels, consultez la page Polymorphisme.

Résumé des interfaces

Une interface possède les propriétés suivantes :

- Une interface est généralement similaire à une classe de base abstraite avec des membres abstraits uniquement. Toute classe ou tout struct qui implémentent l'interface doivent implémenter tous ses membres. Éventuellement, une interface peut définir des implémentations par défaut pour tout ou partie de ses membres. Pour plus d'informations, consultez méthodes d'interface par défaut.
- Une interface ne peut pas être instanciée directement. Ses membres sont implémentées par une classe ou un struct qui implémentent l'interface.
- Une classe ou un struct peuvent implémenter plusieurs interfaces. Une classe peut hériter d'une classe de base et également implémenter une ou plusieurs interfaces.

Sections connexes

- Propriétés de l'interface
- Indexeurs dans les interfaces

- Comment implémenter des événements d'interface
- Classes et structs
- Héritage
- Interfaces
- Méthodes
- Polymorphisme
- Classes abstract et sealed et membres de classe
- Propriétés
- Événements
- Indexeurs

Voir aussi

- Guide de programmation C#
- Héritage
- Noms d'identificateurs

Méthodes dans (C#)

16/12/2020 • 37 minutes to read • Edit Online

Une méthode est un bloc de code qui contient une série d'instructions. Un programme provoque l'exécution des instructions en appelant la méthode et en spécifiant les éventuels arguments de méthode requis. En C#, chaque instruction exécutée est effectuée dans le contexte d'une méthode. La méthode `Main` est le point d'entrée de chaque application C# et elle est appelée par le Common Language Runtime (CLR) au démarrage du programme.

> **NOTE**
> Cette rubrique décrit les méthodes nommées. Pour plus d'informations sur les fonctions anonymes, consultez Fonctions anonymes.

Signatures de méthode

Les méthodes sont déclarées dans une `class` ou une `struct` en spécifiant :

- Un niveau d'accès facultatif, comme `public` ou `private`. La valeur par défaut est `private`.
- Des modificateurs facultatifs, comme `abstract` ou `sealed`.
- La valeur de retour, ou `void` si la méthode n'en a pas.
- Nom de la méthode.
- Des paramètres de méthodes. Les paramètres de méthode sont placés entre parenthèses et séparés par des virgules. Des parenthèses vides indiquent que la méthode ne requiert aucun paramètre.

Ces parties forment ensemble la signature de la méthode.

> **IMPORTANT**
> Un type de retour d'une méthode ne fait pas partie de la signature de la méthode à des fins de surcharge de méthode. Toutefois, il fait partie de la signature de la méthode lors de la détermination de la compatibilité entre un délégué et la méthode vers laquelle il pointe.

L'exemple suivant définit une classe nommée `Motorcycle` qui contient cinq méthodes :

```
using System;

abstract class Motorcycle
{
   // Anyone can call this.
   public void StartEngine() {/* Method statements here */ }

   // Only derived classes can call this.
   protected void AddGas(int gallons) { /* Method statements here */ }

   // Derived classes can override the base class implementation.
   public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

   // Derived classes can override the base class implementation.
   public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

   // Derived classes must implement this.
   public abstract double GetTopSpeed();
}
```

Notez que la classe `Motorcycle` inclut une méthode surchargée, `Drive`. Deux méthodes ont le même nom, mais doivent être différenciées par leurs types de paramètre.

Appel de méthode

Les méthodes peuvent être *d'instance* ou *statiques*. L'appel d'une méthode d'instance nécessite l'instanciation d'un objet et l'appel de la méthode sur cet objet ; une méthode d'instance agit sur cette instance et sur ses données. Vous appelez une méthode statique en référençant le nom du type auquel la méthode appartient ; les méthodes statiques n'agissent pas sur les données d'une instance. Une tentative d'appeler une méthode statique via une instance d'objet génère une erreur du compilateur.

Appeler une méthode est comme accéder à un champ. Après le nom d'objet (si vous appelez une méthode d'instance) ou le nom de type (si vous appelez une méthode `static`), ajoutez un point, le nom de la méthode et des parenthèses. Les arguments sont répertoriés entre parenthèses et séparés par des virgules.

La définition de la méthode spécifie les noms et types des paramètres requis. Quand un appelant appelle la méthode, il fournit des valeurs réelles, appelées « arguments », pour chaque paramètre. Les arguments doivent être compatibles avec le type de paramètre, mais le nom de l'argument, si un nom est utilisé dans le code appelant, ne doit pas nécessairement être le même que celui du paramètre défini dans la méthode. Dans l'exemple suivant, la méthode `Square` comprend un seul paramètre de type `int` nommé *i*. Le premier appel de la méthode passe à la méthode `Square` une variable de type `int` nommée *num*, le deuxième appel passe une constante numérique et le troisième appel passe une expression.

```
public class Example
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

La forme la plus courante des appels de méthode utilise des arguments de position ; elle fournit les arguments dans le même ordre que celui des paramètres de la méthode. Les méthodes de la classe `Motorcycle` peuvent donc être appelées comme dans l'exemple suivant. Par exemple, l'appel à la méthode `Drive` comprend deux arguments qui correspondent aux deux paramètres de la syntaxe de la méthode. Le premier devient la valeur du paramètre `miles` et le deuxième devient la valeur du paramètre `speed`.

```
class TestMotorcycle : Motorcycle
{
   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();

      moto.StartEngine();
      moto.AddGas(15);
      moto.Drive(5, 20);
      double speed = moto.GetTopSpeed();
      Console.WriteLine("My top speed is {0}", speed);
   }
}
```

Vous pouvez également utiliser des *arguments nommés* au lieu des arguments positionnels lors de l'appel d'une méthode. Lors de l'utilisation d'arguments nommés, vous spécifiez le nom du paramètre suivi de deux-points (« : ») et de l'argument. Les arguments de la méthode peuvent apparaître dans n'importe quel ordre, dès lors que tous les arguments nécessaires sont présents. L'exemple suivant utilise des arguments nommés pour appeler la méthode `TestMotorcycle.Drive`. Dans cet exemple, les arguments nommés sont passés dans l'ordre inverse de celui de la liste des paramètres de la méthode.

```
using System;

class TestMotorcycle : Motorcycle
{
   public override int Drive(int miles, int speed)
   {
      return (int) Math.Round( ((double)miles) / speed, 0);
   }

   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();
      moto.StartEngine();
      moto.AddGas(15);
      var travelTime = moto.Drive(speed: 60, miles: 170);
      Console.WriteLine("Travel time: approx. {0} hours", travelTime);
   }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Vous pouvez appeler une méthode en utilisant à la fois des arguments positionnels et des arguments nommés. Toutefois, les arguments positionnels peuvent uniquement suivre des arguments nommés lorsque les arguments nommés sont dans les positions appropriées. L'exemple suivant appelle la méthode `TestMotorcycle.Drive` de l'exemple précédent en utilisant un argument positionnel et un argument nommé.

```
var travelTime = moto.Drive(170, speed: 55);
```

Méthodes hérités et remplacées

En plus des membres qui sont définis explicitement dans un type, un type hérite des membres définis dans ses classes de base. Comme tous les types du système de types managés héritent directement ou indirectement de la classe Object, tous les types héritent ses membres, comme Equals(Object), GetType() et ToString(). L'exemple suivant définit une classe `Person`, instancie deux objets `Person` et appelle la méthode `Person.Equals` pour déterminer si les deux objets sont égaux. La méthode `Equals` n'est cependant pas définie dans la classe `Person` ; elle est héritée de Object.

```
using System;

public class Person
{
   public String FirstName;
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: False
```

Vous pouvez remplacer des membres hérités par des types en utilisant le mot clé `override` et en fournissant une implémentation de la méthode remplacée. La signature de la méthode doit être identique à celle de la méthode remplacée. L'exemple suivant est semblable au précédent, sauf qu'il remplace la méthode Equals(Object). (Il remplace aussi la méthode GetHashCode(), car les deux méthodes sont destinées à fournir des résultats cohérents.)

```
using System;

public class Person
{
   public String FirstName;

   public override bool Equals(object obj)
   {
      var p2 = obj as Person;
      if (p2 == null)
         return false;
      else
         return FirstName.Equals(p2.FirstName);
   }

   public override int GetHashCode()
   {
      return FirstName.GetHashCode();
   }
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: True
```

Passage de paramètres

Les types en C# sont des *types valeur* ou des *types référence*. Pour obtenir la liste des types valeur intégrés, consultez types. Par défaut, les types valeur et les types référence sont passés par valeur à une méthode.

Passage de paramètres par valeur

Quand un type valeur est passé par valeur à une méthode, c'est une copie de l'objet, et non pas l'objet lui-même, qui est passée à la méthode. Par conséquent, les modifications de l'objet dans la méthode appelée n'ont pas d'effet sur l'objet d'origine quand le contrôle retourne à l'appelant.

L'exemple suivant passe un type valeur à une méthode par valeur, et la méthode appelée tente de changer la valeur du type valeur. Il définit une variable de type `int`, qui est un type valeur, initialise sa valeur à 20 et la passe à une méthode nommée `ModifyValue` qui change la valeur de la variable en 30. Quand la méthode retourne, la valeur de la variable reste cependant inchangée.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Quand un objet d'un type référence est passé par valeur à une méthode, c'est une référence à l'objet qui est passée par valeur. Autrement dit, la méthode ne reçoit pas l'objet lui-même mais un argument qui indique l'emplacement de l'objet. Si vous modifiez un membre de l'objet en utilisant cette référence, la modification est répercutée dans l'objet quand le contrôle retourne à la méthode appelante. Cependant, remplacer l'objet passé à la méthode n'a pas d'effet sur l'objet d'origine quand le contrôle retourne à l'appelant.

L'exemple suivant définit une classe (qui est un type référence) nommée `SampleRefType`. Il instancie un objet `SampleRefType`, affecte la valeur 44 à son champ `value` et passe l'objet à la méthode `ModifyObject`. L'exemple fait essentiellement la même chose que l'exemple précédent : il passe un argument par valeur à une méthode. Cependant, comme il utilise un type référence, le résultat est différent. La modification apportée dans `ModifyObject` au champ `obj.value` change également en 33 le champ `value` de l'argument `rt` dans la méthode `Main`, comme le montre le résultat de l'exemple.

```
using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Passage de paramètres par référence

On passe un paramètre par référence quand on souhaite changer la valeur d'un argument dans une méthode et refléter cette modification lorsque le contrôle revient à la méthode appelante. Pour passer un paramètre par référence, vous utilisez le `ref` `out` mot clé ou. Vous pouvez également passer une valeur par référence pour éviter de copier tout en empêchant les modifications à l'aide du `in` mot clé.

L'exemple suivant est identique au précédent, sauf que la valeur est passée par référence à la méthode `ModifyValue`. Quand la valeur du paramètre est modifiée dans la méthode `ModifyValue`, le changement de la valeur est reflété quand le contrôle retourne à l'appelant.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Un modèle courant qui utilise des paramètres par référence implique la permutation des valeurs des variables. Vous passez deux variables à une méthode par référence, et la méthode permute leur contenu. L'exemple suivant permute des valeurs d'entier.

```
using System;

public class Example
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passer un paramètre de type référence vous permet de changer la valeur de la référence elle-même, au lieu de la valeur de ses éléments ou champs individuels.

Tableaux de paramètres

Parfois, l'obligation de spécifier le nombre exact d'arguments de votre méthode est restrictive. En utilisant le mot clé `params` pour indiquer qu'un paramètre est un tableau de paramètres, vous permettez que votre méthode soit appelée avec un nombre variable d'arguments. Le paramètre marqué avec le mot clé `params` doit être un type tableau. Ce doit être le dernier paramètre de la liste de paramètres de la méthode.

Un appelant peut ensuite appeler la méthode de l'une des quatre manières suivantes :

- En passant un tableau du type approprié, qui contient le nombre d'éléments souhaité.
- En passant une liste d'arguments individuels séparés par des virgules et du type approprié à la méthode.
- En passant `null` .
- En ne fournissant pas d'argument au tableau de paramètres.

L'exemple suivant définit une méthode nommée `GetVowels` qui retourne toutes les voyelles à partir d'un tableau de paramètres. La `Main` méthode illustre les quatre façons d'appeler la méthode. Les appelants n'ont pas nécessairement à fournir tous les arguments pour les paramètres qui incluent le modificateur `params`. Dans ce cas, le paramètre est un tableau vide.

```
using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Paramètres et arguments facultatifs

Une définition de méthode peut spécifier que ses paramètres sont obligatoires ou facultatifs. Par défaut, les paramètres sont obligatoires. Vous spécifiez des paramètres facultatifs en incluant la valeur par défaut du paramètre dans la définition de la méthode. Quand la méthode est appelée, si aucun argument n'est fourni pour un paramètre facultatif, c'est la valeur par défaut qui est utilisée à la place.

La valeur par défaut du paramètre doit être affectée par un des types d'expressions suivants :

- Une constante, comme une chaîne littérale ou un nombre.
- Une expression de la forme `new ValType()`, où `ValType` est un type valeur. Notez que ceci appelle le constructeur sans paramètre implicite du type valeur, qui n'est pas un membre réel du type.
- Une expression de la forme `default(ValType)`, où `ValType` est un type valeur.

Si une méthode comprend à la fois des paramètres obligatoires et des paramètres facultatifs, les paramètres facultatifs sont définis à la fin de la liste des paramètres, après tous les paramètres obligatoires.

L'exemple suivant définit une méthode, `ExampleMethod`, qui a un paramètre obligatoire et deux paramètres facultatifs.

```
using System;

public class Options
{
   public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
   {
      Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                        optionalInt, required + optionalInt);
   }
}
```

Si une méthode avec plusieurs arguments facultatifs est appelée en utilisant des arguments positionnels, l'appelant doit fournir un argument pour tous les paramètres facultatifs, du premier au dernier pour lequel un argument est fourni. Par exemple, dans le cas de la méthode `ExampleMethod`, si l'appelant fournit un argument pour le paramètre `description`, il doit également en fournir un pour le paramètre `optionalInt`.
`opt.ExampleMethod(2, 2, "Addition of 2 and 2");` est un appel de méthode valide.
`opt.ExampleMethod(2, , "Addition of 2 and 0");` génère une erreur du compilateur « Argument manquant ».

Si une méthode est appelée en utilisant des arguments nommés ou une combinaison d'arguments positionnels et nommés, l'appelant peut omettre les arguments qui suivent le dernier argument positionnel dans l'appel de la méthode.

L'exemple suivant appelle la méthode `ExampleMethod` trois fois. Les deux premiers appels de la méthode utilisent des arguments positionnels. Le premier omet les deux arguments facultatifs, tandis que le deuxième omet le dernier argument. Le troisième appel de méthode fournit un argument positionnel pour le paramètre requis, mais utilise un argument nommé pour fournir une valeur au `description` paramètre tout en omettant l' `optionalInt` argument.

```
public class Example
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

L'utilisation de paramètres facultatifs affecte la *résolution de la surcharge*, qui est la façon dont le compilateur C# détermine quelle surcharge particulière doit être appelée par un appel de méthode, comme suit :

- Une méthode, un indexeur ou un constructeur est un candidat pour l'exécution si chacun de ses paramètres est facultatif ou correspond, par nom ou par position, à un seul argument dans l'instruction appelante, et que cet argument peut être converti vers le type du paramètre.
- Si plusieurs candidats sont trouvés, les règles de résolution de surcharge des conversions préférées sont appliquées aux arguments qui sont explicitement spécifiés. Les arguments omis pour les paramètres facultatifs sont ignorés.
- Si deux candidats sont jugés de qualité équivalente, la préférence va à celui qui n'a pas de paramètres facultatifs pour lesquels des arguments ont été omis dans l'appel. Ceci est une conséquence d'une préférence générale dans la résolution de la surcharge pour les candidats qui ont le moins de paramètres.

Valeurs retournées

Les méthodes peuvent retourner une valeur à l'appelant. Si le type de retour (le type qui figure avant le nom de la méthode) n'est pas `void`, la méthode peut retourner la valeur en utilisant le mot clé `return`. Une instruction avec le mot clé `return` suivi d'une variable, d'une constante ou d'une expression qui correspond au type de retour retourne cette valeur à l'appelant de la méthode. Les méthodes dotées d'un type de retour non void doivent utiliser le mot clé `return` pour retourner une valeur. Le mot clé `return` arrête également l'exécution de la méthode.

Si le type de retour est `void`, une instruction `return` sans valeur est quand même utile pour arrêter l'exécution de la méthode. Sans le mot clé `return`, la méthode arrête de s'exécuter quand elle atteint la fin du bloc de code.

Par exemple, ces deux méthodes utilisent le mot clé `return` pour retourner des entiers :

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Pour utiliser une valeur retournée à partir d'une méthode, la méthode d'appel peut utiliser l'appel de méthode proprement dit partout où une valeur du même type peut suffire. Vous pouvez également affecter la valeur de retour à une variable. Par exemple, les deux exemples de code suivants remplissent le même objectif :

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

L'utilisation d'une variable locale, dans cet exemple, `result`, pour stocker une valeur est facultative. Elle peut favoriser la lisibilité du code ou s'avérer nécessaire si vous avez besoin de stocker la valeur d'origine de l'argument pour la portée entière de la méthode.

Vous voulez parfois que votre méthode retourne plusieurs valeurs. À compter de C# 7.0, vous pouvez faire cela facilement en utilisant des *types tuple* et des *littéraux de tuple*. Le type tuple définit les types de données des éléments du tuple. Les littéraux de tuple fournissent les valeurs réelles du tuple retourné. Dans l'exemple suivant, `(string, string, string, int)` définit le type tuple retourné par la méthode `GetPersonalInfo`. L'expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` est le littéral de tuple ; la méthode retourne le prénom, le deuxième prénom et le nom, ainsi que l'âge pour un objet `PersonInfo`.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

L'appelant peut alors consommer le tuple retourné avec du code comme celui-ci :

```
var person = GetPersonalInfo("111111111")
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Vous pouvez aussi affecter des noms aux éléments du tuple dans la définition du type tuple. L'exemple suivant montre une autre version de la méthode `GetPersonalInfo` qui utilise des éléments nommés :

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

L'appel précédent de la méthode `GetPersonInfo` peut alors être modifié comme suit :

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Si un tableau est passé comme argument à une méthode et que celle-ci modifie la valeur d'éléments individuels, il n'est pas nécessaire que la méthode retourne le tableau, même si vous pouvez choisir de le faire pour la qualité du style ou pour obtenir un flux fonctionnel de valeurs. La raison en est que C# passe tous les types référence par valeur et que la valeur d'une référence de tableau est le pointeur vers ce tableau. Dans l'exemple suivant, les modifications apportées au contenu du tableau `values` qui sont effectuées dans la méthode `DoubleValues` sont observables par tout code ayant une référence au tableau.

```
using System;

public class Example
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Méthodes d'extension

En règle générale, il existe deux façons d'ajouter une méthode à un type existant :

- Modifier le code source pour ce type. Vous ne pouvez évidemment pas le faire si vous ne disposez pas du code source du type. D'autre part, ceci devient une modification d'importance si vous ajoutez aussi des champs de données privés pour prendre en charge la méthode.
- Définir la nouvelle méthode dans une classe dérivée. Vous ne pouvez pas ajouter une méthode de cette façon

en utilisant l'héritage pour d'autres types, comme des structures et des énumérations. Vous ne pouvez pas non plus l'utiliser pour « ajouter » une méthode à une classe scellée.

Les méthodes d'extension vous permettent d'« ajouter » une méthode à un type existant sans modifier le type lui-même, ou en implémentant la nouvelle méthode dans un type hérité. La méthode d'extension ne doit pas non plus nécessairement se trouver dans le même assembly que le type qu'elle étend. Vous appelez une méthode d'extension comme si elle était un membre défini d'un type.

Pour plus d'informations, consultez Méthodes d'extension.

Méthodes async

La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels explicites ni fractionner manuellement votre code entre plusieurs méthodes ou expressions lambda.

Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression `await` dans la méthode async, il retourne à l'appelant si la tâche attendue n'est pas terminée, et la progression dans la méthode avec le mot clé `await` est interrompue jusqu'à ce que la tâche attendue se termine. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode.

> **NOTE**
> Une méthode Async retourne à l'appelant lorsqu'elle rencontre le premier objet attendu qui n'est pas encore terminé ou qu'elle atteint la fin de la méthode Async, selon ce qui se produit en premier.

Une méthode async peut avoir un type de retour Task<TResult>, Task ou `void`. Le type de retour `void` est essentiellement utilisé pour définir les gestionnaires d'événements, où un type de retour `void` est obligatoire. Une méthode async qui retourne `void` ne peut pas être attendue, et l'appelant d'une méthode retournant void ne peut intercepter aucune exception levée par la méthode. À compter de C# 7.0, une méthode asynchrone peut avoir n'importe quel type de retour de type tâche.

Dans l'exemple suivant, `DelayAsync` est une méthode async contenant une instruction return qui retourne un entier. Comme il s'agit d'une méthode async, la déclaration de sa méthode doit avoir un type de retour `Task<int>`. Comme le type de retour est `Task<int>`, l'évaluation de l'expression `await` dans `DoSomethingAsync` produit un entier, comme l'instruction `int result = await delayTask` suivante le montre.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Une méthode async ne peut pas déclarer de paramètres in, ref ou out, mais elle peut appeler des méthodes qui ont ces paramètres.

Pour plus d'informations sur les méthodes Async, consultez programmation asynchrone avec Async et await et Async (types de retour).

Membres expression-bodied

Il est courant d'avoir des définitions de méthode qui retournent tout simplement le résultat d'une expression immédiatement, ou qui ont une seule instruction en tant que corps de la méthode. Il existe un raccourci de syntaxe pour définir de telles méthodes en utilisant `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si la méthode retourne `void` ou est une méthode async, alors le corps de la méthode doit être une expression d'instruction (comme avec les expressions lambda). Les propriétés et les indexeurs doivent être en lecture seule, et vous n'y utilisez pas le mot clé d'accesseur `get`.

Iterators

Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actif est mémorisé pour que l'appelant puisse demander l'élément suivant dans la séquence.

Le type de retour d'un itérateur peut être IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Pour plus d'informations, consultez itérateurs.

Voir aussi

- Modificateurs d'accès
- Classes statiques et membres de classe statique
- Héritage
- Classes abstract et sealed et membres de classe
- params
- à
- ref
- in
- Passer des paramètres

Propriétés

16/12/2020 • 16 minutes to read • Edit Online

Les propriétés sont des éléments de première classe dans C#. Le langage définit la syntaxe que les développeurs utilisent pour écrire du code qui exprime leur intention de conception avec précision.

Les propriétés auxquelles le code accède se comportent comme des champs. Toutefois, contrairement aux champs, les propriétés sont implémentées avec des accesseurs qui définissent quelles instructions sont exécutées au moment de l'accès à une propriété ou de son assignation.

Syntaxe des propriétés

La syntaxe des propriétés est une extension naturelle des champs. Un champ définit un emplacement de stockage :

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

Une définition de propriété contient les déclarations de l'accesseur `get`, qui récupère la valeur de cette propriété, et de l'accesseur `set`, qui assigne cette valeur :

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

La syntaxe illustrée ci-dessus est la syntaxe *auto property*. Le compilateur génère l'emplacement de stockage pour le champ qui enregistre la propriété. Le compilateur implémente également le corps des accesseurs `get` et `set`.

Parfois, vous devez initialiser une propriété sur une valeur autre que la valeur par défaut pour son type. C# permet cette opération en définissant une valeur après l'accolade fermante de la propriété. Vous pouvez choisir comme valeur initiale pour la propriété `FirstName` une chaîne vide au lieu de `null`. Vous pouvez le spécifier comme indiqué ci-dessous :

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

Une initialisation spécifique est pratique surtout pour les propriétés en lecture seule, comme vous le verrez plus loin dans cet article.

Vous pouvez aussi définir le stockage vous-même, de la manière suivante :

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Quand une implémentation de propriété est une expression unique, vous pouvez utiliser des *membres expression-bodied* pour l'accesseur Get ou Set :

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Cette syntaxe simplifiée est utilisée partout où elle est applicable dans cet article.

La définition de propriété présentée ci-dessus est une propriété en lecture-écriture. Notez la présence du mot clé `value` dans l'accesseur set. L'accesseur `set` a toujours un seul paramètre nommé `value`. L'accesseur `get` doit retourner une valeur convertible dans le type de la propriété (`string`, dans cet exemple).

Nous venons de voir les éléments de base de la syntaxe. Il existe beaucoup de variantes de cette syntaxe, qui sont adaptées aux différents idiomes de conception. Nous allons les explorer et découvrir les options syntaxiques de chacune.

Scénarios

Les exemples ci-dessus ont montré un des cas les plus simples de définition de propriété, à savoir une propriété en lecture-écriture sans validation. En écrivant le code souhaité dans les accesseurs `get` et `set`, vous pouvez créer de nombreux scénarios différents.

Validation

Vous pouvez écrire du code dans l'accesseur `set` pour garantir que les valeurs représentées par une propriété sont toujours valides. Par exemple, définissez une règle pour la classe `Person` qui spécifie que le nom ne peut pas être vide, ni contenir d'espace blanc. Le code à écrire est le suivant :

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

L'exemple précédent peut être simplifié à l'aide d'une `throw` expression dans le cadre de la validation de l'accesseur Set de la propriété :

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

L'exemple ci-dessus applique la règle selon laquelle le nom ne doit pas être vide, ni contenir d'espace blanc. Supposons qu'un développeur écrive cette ligne de code :

```
hero.FirstName = "";
```

Cette assignation lève une exception `ArgumentException`. Étant donné qu'un accesseur set de propriété doit avoir un type de retour void, vous signalez les erreurs dans l'accesseur set en levant une exception.

Vous pouvez employer cette même syntaxe pour valider d'autres éléments dans votre scénario. Vous pouvez notamment vérifier les relations entre plusieurs propriétés ou effectuer une validation par rapport à des conditions externes. Toute instruction C# valide peut être utilisée dans un accesseur de propriété.

Lecture seule

Jusqu'ici, nous avons vu uniquement des définitions de propriétés qui sont en lecture-écriture dans des accesseurs publics. Ce n'est pas la seule accessibilité valide pour les propriétés. Vous pouvez créer des propriétés en lecture seule, ou assigner une accessibilité différente aux accesseurs set et get. Supposons que votre classe `Person` doit uniquement autoriser la modification de la valeur de la propriété `FirstName` à partir des autres méthodes de cette classe. Vous pouvez alors assigner l'accessibilité `private` au lieu de `public` à l'accesseur set :

```
public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}
```

À présent, la propriété `FirstName` est accessible à partir de n'importe quel code, mais elle peut uniquement être assignée à partir de code dans la classe `Person`.

Vous pouvez ajouter n'importe quel modificateur d'accès restrictif à l'accesseur set ou get. Le modificateur d'accès que vous ajoutez à un accesseur doit être plus restrictif que le modificateur d'accès spécifié dans la définition de propriété. Le code ci-dessus est autorisé, car la propriété `FirstName` est `public`, mais l'accesseur set est `private`. En revanche, vous ne pouvez pas déclarer une propriété `private` avec un accesseur `public`. Les propriétés peuvent également être déclarées comme `protected`, `internal`, `protected internal` ou même `private`.

Placer le modificateur le plus restrictif sur l'accesseur `get` est également autorisé. Par exemple, vous pouvez avoir une propriété `public`, mais restreindre l'accesseur `get` à `private`. Ce scénario s'observe rarement dans la pratique.

Vous pouvez également limiter les modifications apportées à une propriété pour qu'elle puisse uniquement être définie dans un constructeur ou un initialiseur de propriété. Vous pouvez modifier la classe `Person`, comme suit :

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

Cette fonctionnalité est généralement utilisée pour initialiser des collections qui sont exposées sous forme de propriétés en lecture seule :

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

Propriétés calculées

Une propriété peut faire plus que simplement retourner la valeur d'un champ de membre. Vous pouvez créer des propriétés qui retournent une valeur calculée. L'objet `Person` est étendu pour retourner le nom complet, calculé en concaténant le nom et le prénom :

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

L'exemple ci-dessus utilise la fonctionnalité d'interpolation de chaîne pour créer la chaîne mise en forme du nom complet.

Vous pouvez également utiliser un *membre expression-bodied*, qui constitue un moyen plus succinct de créer la propriété `FullName` calculée :

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Les *membres expression-bodied* utilisent la syntaxe des *expressions lambda* pour définir des méthodes qui contiennent une seule expression. Ici, cette expression retourne le nom complet de l'objet person.

Propriétés évaluées avec mise en cache

Vous pouvez combiner le concept d'une propriété calculée avec le stockage et créer une *propriété évaluée avec mise en cache*. Par exemple, vous pouvez mettre à jour la propriété `FullName` pour que la chaîne soit mise en forme uniquement lors du premier accès à cette propriété :

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Le code ci-dessus contient toutefois un bogue. Si le code met à jour la valeur de la propriété `FirstName` ou `LastName`, le champ `fullName` qui a été précédemment évalué n'est plus valide. Vous modifiez les accesseurs `set` des propriétés `FirstName` et `LastName` pour que le champ `fullName` soit recalculé :

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Dans cette version finale du code, la propriété `FullName` est évaluée uniquement si cela est nécessaire. Si la version précédemment calculée est valide, elle est utilisée. Si elle n'est plus valide en raison d'un changement d'état, la version est recalculée. Les développeurs peuvent utiliser cette classe sans connaître les détails de l'implémentation. Ces modifications internes n'ont pas d'impact sur l'utilisation de l'objet Person. C'est l'un des principaux avantages d'utiliser des propriétés pour exposer les membres de données d'un objet.

Attachement d'attributs à des propriétés implémentées automatiquement

À compter de C# 7.3, les attributs de champ peuvent être attachés au champ de stockage généré par le compilateur dans les propriétés implémentées automatiquement. Par exemple, considérez une révision de la classe `Person` qui ajoute une propriété `Id` unique de type entier. Vous écrivez la `Id` propriété à l'aide d'une propriété implémentée automatiquement, mais votre conception n'appelle pas pour rendre la propriété persistante `Id` . NonSerializedAttribute peut être attaché seulement à des champs, et pas à des propriétés. Vous pouvez attacher NonSerializedAttribute au champ de stockage pour la propriété `Id` en utilisant le spécificateur `field:` sur l'attribut, comme illustré dans l'exemple suivant :

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Cette technique fonctionne pour tout attribut que vous voulez attacher au champ de stockage sur la propriété implémentée automatiquement.

Implémentation de INotifyPropertyChanged

Il existe un dernier scénario où vous devrez écrire du code dans un accesseur de propriété : pour prendre en charge l'interface INotifyPropertyChanged, qui notifie les changements de valeurs aux clients de liaison de données. Quand la valeur d'une propriété change, l'objet déclenche l'événement INotifyPropertyChanged.PropertyChanged pour signaler le changement. Les bibliothèques de liaison de données mettent ensuite à jour les éléments d'affichage en fonction de cette modification. Le code ci-dessous montre comment implémenter `INotifyPropertyChanged` pour la propriété `FirstName` de la classe Person.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

L'opérateur `?.` est appelé *opérateur conditionnel Null*. Il recherche une référence null avant d'évaluer le côté droit de l'opérateur. Au final, s'il n'y a pas d'abonné à l'événement `PropertyChanged`, le code devant déclencher l'événement n'est pas exécuté. Dans ce cas précis, il lèverait une exception `NullReferenceException` sans cette vérification. Pour plus d'informations, consultez `events`. Cet exemple utilise également le nouvel opérateur `nameof` pour convertir le symbole de nom de propriété en sa représentation textuelle. L'utilisation de `nameof` peut vous éviter des erreurs dues à la saisie incorrecte du nom de propriété.

L'implémentation de INotifyPropertyChanged est un autre exemple de cas où vous pouvez écrire du code dans vos accesseurs pour prendre en charge les scénarios souhaités.

Résumé

Les propriétés sont une forme de champs intelligents dans une classe ou un objet. De l'extérieur de l'objet, elles

apparaissent sous la forme de champs dans l'objet. Toutefois, les propriétés peuvent être implémentées avec toutes les fonctionnalités C#. Vous pouvez écrire du code qui remplit les exigences de validation, d'accessibilité, d'évaluation différée ou toute autre exigence requise dans vos scénarios.

Indexeurs

02/11/2020 • 15 minutes to read • Edit Online

Les *indexeurs* sont similaires aux propriétés. Les indexeurs sont souvent construits à l'aide des mêmes fonctionnalités de langage que les propriétés. Il permettent l'utilisation de propriétés *indexées*, qui sont référencées à l'aide d'un ou plusieurs arguments. Ces arguments fournissent un index dans une collection de valeurs.

Syntaxe de l'indexeur

Vous pouvez accéder à un indexeur avec un nom de variable et des crochets. Vous devez placer les arguments de l'indexeur entre crochets :

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

Déclarez les indexeurs en utilisant le mot clé `this` comme nom de propriété et en déclarant les arguments entre crochets. Cette déclaration correspond à l'utilisation abordée dans le paragraphe précédent :

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

Dans ce premier exemple, vous pouvez voir la relation entre la syntaxe des propriétés et celle des indexeurs. Cette analogie s'applique à la plupart des règles de syntaxe des indexeurs. Les indexeurs peuvent avoir n'importe quel modificateur d'accès valide (public, protected internal, protected, internal, private ou private protected). Ils peuvent être sealed, virtual ou abstract. Comme pour les propriétés, vous pouvez spécifier des modificateurs d'accès différents pour les accesseurs get et set d'un indexeur. Vous pouvez également spécifier des indexeurs en lecture seule (en omettant l'accesseur set) ou des indexeurs en écriture seule (en omettant l'accesseur get).

Vous pouvez appliquer aux indexeurs quasiment tout ce que vous avez appris de l'utilisation des propriétés. La seule exception à cette règle sont les *propriétés implémentées automatiquement*. Le compilateur ne peut pas toujours générer le stockage adapté à l'indexeur.

C'est la présence d'arguments référençant un élément dans un ensemble d'éléments qui distingue les indexeurs des propriétés. Vous pouvez définir plusieurs indexeurs sur un type, tant que chaque indexeur a sa propre liste d'arguments. Nous allons explorer différents scénarios où vous pourrez utiliser un ou plusieurs indexeurs dans une définition de classe.

Scénarios

Vous définissez des *indexeurs* dans votre type quand son API modélise une collection dans laquelle vous définissez des arguments. Vos indexeurs peuvent ou non être mappés directement aux types de collection qui font partie du framework .NET Core. Votre type peut avoir d'autres responsabilités en plus de la modélisation d'une collection. Les indexeurs vous permettent de fournir l'API qui correspond à l'abstraction de votre type sans exposer les détails internes concernant le stockage ou le calcul des valeurs de cette abstraction.

Examinons quelques-uns des scénarios courants d'utilisation des *indexeurs*. Vous pouvez accéder au dossier d'exemples d'indexeurs. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Tableaux et vecteurs

L'un des scénarios de création d'indexeur les plus courants est lorsque votre type modélise un tableau ou un vecteur. Vous pouvez créer un indexeur pour modéliser une liste de données triées.

L'avantage de créer votre propre indexeur est que vous pouvez définir le stockage de cette collection en fonction de vos besoins. Imaginez un scénario où votre type modélise des données d'historique qui sont trop volumineuses pour être chargées en une seule fois dans la mémoire. Vous devez charger et décharger des sections de la collection selon leur utilisation. L'exemple suivant modélise ce comportement. Il signale le nombre de points de données. Il crée à la demande des pages contenant des sections de données. Il supprime des pages de la mémoire afin de libérer de l'espace pour les pages qui ont fait l'objet de demandes récentes.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }

    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();
```

```
    public DataSamples(int totalSize)
    {
        this.totalSize = totalSize;
    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}
```

Vous pouvez suivre cet idiome de conception afin de modéliser toute sorte de collection pour laquelle il existe de bonnes raisons de ne pas charger l'intégralité des données en mémoire. Notez que la classe `Page` est une classe privée imbriquée qui ne fait pas partie de l'interface publique. Ces détails sont masqués pour tous les utilisateurs de cette classe.

Dictionnaires

Un autre scénario courant est lorsque vous avez besoin de modéliser un dictionnaire ou un mappage. Dans ce cas, votre type stocke des valeurs en fonction des clés, généralement des clés de texte. Cet exemple crée un dictionnaire qui mappe les arguments de ligne de commande à des expressions lambda qui gèrent ces options. L'exemple suivant montre deux classes : une classe `ArgsActions` qui mappe une option de ligne de commande à un délégué `Action`, et un `ArgsProcessor` qui utilise `ArgsActions` pour exécuter chaque `Action` quand il rencontre cette option.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

Dans cet exemple, la collection `ArgsAction` correspond étroitement à la collection sous-jacente. `get` détermine si une option donnée a été configurée. Si c'est le cas, il retourne le `Action` associé à cette option. Sinon, il retourne un `Action` qui n'a aucun effet. L'accesseur public n'inclut pas d'accesseur `set`, mais il comprend la conception qui utilise une méthode publique pour définir des options.

Mappages multidimensionnels

Vous pouvez créer des indexeurs qui utilisent plusieurs arguments. En outre, ces arguments ne sont pas contraints à être du même type. Examinons ces deux exemples.

Le premier exemple montre une classe qui génère des valeurs pour un ensemble de Mandelbrot. Pour plus d'informations sur les mathématiques impliquées par cet ensemble, lisez cet article. L'indexeur utilise deux doubles pour définir un point dans le plan X, Y. L'accesseur get calcule le nombre d'itérations jusqu'à un point déterminé pour ne pas se trouver dans l'ensemble. Si le nombre maximal d'itérations est atteint, le point se trouve dans l'ensemble, et la valeur maxIterations de la classe est retournée. Les images générées par ordinateur popularisées par l'ensemble de Mandelbrot définissent des couleurs pour le nombre d'itérations qui sont nécessaires pour

déterminer qu'un point se trouve en dehors de l'ensemble.

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

L'ensemble de Mandelbrot définit des valeurs à chaque coordonnée (x, y) pour des valeurs de nombres réels. Cela définit un dictionnaire qui peut contenir un nombre infini de valeurs. Par conséquent, aucun stockage n'est nécessaire. Cette classe calcule la valeur de chaque point lorsque le code appelle l'accesseur `get`. Aucun stockage sous-jacent n'est utilisé.

Examinons une dernière utilisation d'indexeur, dans laquelle l'indexeur accepte plusieurs arguments de types différents. Prenons un programme qui gère des données d'historique des températures. Cet indexeur utilise une ville et une date pour définir ou obtenir les températures minimales et maximales de la ville en question :

```
using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}
```

Cet exemple crée un indexeur qui mappe des données météorologiques sur deux arguments différents : une ville (représentée par un `string`) et une date (représentée par un `DateTime`). Le stockage interne utilise deux classes `Dictionary` pour représenter le dictionnaire à deux dimensions. L'API publique ne représente plus le stockage sous-jacent. Les fonctionnalités de langage des indexeurs vous permettent de créer une interface publique qui représente votre abstraction, même si le stockage sous-jacent doit utiliser des types de collections de base différents.

Ce code comprend deux sections avec lesquelles vous pouvez ne pas être familier. Ces deux `using` directives :

```
using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;
```

créent un *alias* pour un type générique construit. Ces instructions permettent au code d'utiliser plus tard les noms plus descriptifs que sont `DateMeasurements` et `CityDateMeasurements`, au lieu de la construction générique de `Dictionary<DateTime, Measurements>` et `Dictionary<string, Dictionary<DateTime, Measurements> >`. Cette construction nécessite l'utilisation de noms de types qualifiés complets à droite du signe `=`.

La deuxième technique consiste à supprimer les sections de date et heure de tous les objets `DateTime` utilisés pour

indexer des collections. .NET n'inclut pas de type Date uniquement. Les développeurs utilisent le type `DateTime`, mais utilisent la propriété `Date` pour s'assurer qu'il n'existe pas d'objets `DateTime` égaux pour ce jour-là.

Récapitulatif

Vous devez créer des indexeurs chaque fois que vous avez un élément de type propriété dans votre classe, où cette propriété ne représente pas une valeur unique, mais une collection de valeurs dans laquelle chaque élément est identifié par un ensemble d'arguments. Ces arguments peuvent identifier quel élément de la collection doit être référencé. Les indexeurs étendent le concept de Propriétés, où un membre est traité comme un élément de données provenant de l'extérieur de la classe, mais comme une méthode à l'intérieur. Les indexeurs autorisent les arguments à rechercher un élément d'une propriété qui représente un ensemble d'éléments.

Éléments ignorés - Guide C#

16/12/2020 • 12 minutes to read • Edit Online

À compter de C# 7,0, C# prend en charge les éléments ignorés, qui sont des variables d'espace réservé qui sont intentionnellement inutilisées dans le code de l'application. Les éléments ignorés sont équivalents à des variables non affectées : elles n'ont pas de valeur. Comme il n'existe qu'une seule variable d'élément ignoré et qu'un stockage ne peut même pas lui être alloué, les éléments ignorés peuvent réduire les allocations de mémoire. Dans la mesure où elles éclairent l'intention de votre code, elles améliorent sa lisibilité et sa maintenabilité.

Vous indiquez qu'une variable est un élément ignoré en lui affectant comme nom le trait de soulignement (`_`). Par exemple, l'appel de méthode suivant retourne un tuple de 3 éléments dans lequel les première et deuxième valeurs sont ignorées et dans lequel *area* est une variable précédemment déclarée à laquelle vous devez affecter le troisième composant correspondant retourné par *GetCityInformation* :

```
(_, _, area) = city.GetCityInformation(cityName);
```

Dans C# 7,0 et versions ultérieures, les éléments ignorés sont pris en charge dans les affectations dans les contextes suivants :

- Déconstruction d' deconstructionun tuple et d'un objet.
- Critères spéciaux avec is et switch.
- Appels à des méthodes avec des paramètres `out`.
- Un `_` autonome quand aucun `_` n'est dans l'étendue.

À compter de C# 9,0, vous pouvez utiliser des éléments ignorés pour spécifier les paramètres d'entrée inutilisés d'une expression lambda. Pour plus d'informations, consultez la section paramètres d'entrée d'une expression lambda de l'article expressions lambda .

Quand `_` est un élément ignoré valide, une tentative de récupérer sa valeur ou de l'utiliser dans une opération d'affectation génère l'erreur de compilateur CS0301, « Le nom '_' n'existe pas dans le contexte actuel ». La raison en est qu'aucune valeur n'est affectée à `_` et qu'il n'est même pas possible de lui affecter un emplacement de stockage. S'il s'agissait d'une variable réelle, vous ne pourriez pas ignorer plus d'une valeur, comme l'a fait l'exemple précédent.

Déconstruction de tuple et d'objet

Les éléments ignorés sont particulièrement utiles pour travailler avec des tuples quand le code de votre application utilise certains éléments d'un tuple mais ignore les autres. Par exemple, la méthode `QueryCityDataForYears` suivante retourne un tuple de 6 éléments avec le nom d'une ville, sa région, une année, la population de la ville pour cette année, une seconde année et la population de la ville pour cette seconde année. L'exemple montre la différence de population entre ces deux années. Parmi les données disponibles dans le tuple, nous ne sommes pas intéressés par la région de la ville, et nous connaissons le nom de la ville et les deux dates au moment du design. Par conséquent, nous sommes intéressés seulement par les deux valeurs de la population stockées dans le tuple et nous pouvons gérer ses valeurs restantes comme éléments ignorés.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Pour plus d'informations sur la déconstruction de tuples avec des éléments ignorés, consultez Déconstruction de tuples et d'autres types.

La méthode `Deconstruct` d'une classe, d'un struct ou d'une interface vous permet aussi de récupérer et de déconstruire un ensemble spécifique de données d'un objet. Vous pouvez utiliser des éléments ignorés quand vous êtes intéressé seulement par un sous-ensemble des valeurs déconstruites. L'exemple suivant déconstruit un objet `Person` en quatre chaînes (le prénom, le nom, la ville et l'État), mais ignore le nom et l'État.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // <Snippet1>
        // Deconstruct the person object.
        var (fName, _, city, _) = p;
        Console.WriteLine($"Hello {fName} of {city}!");
        // The example displays the following output:
        //      Hello John of Boston!
        // </Snippet1>
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Pour plus d'informations sur la déconstruction de types définis par l'utilisateur avec des éléments ignorés, consultez Déconstruction de tuples et d'autres types.

Critères spéciaux avec `switch` et `is`

Le *modèle d'élément ignoré* peut être utilisé dans des critères spéciaux avec les mots clés is et switch. Chaque expression correspond toujours au modèle d'élément ignoré.

L'exemple suivant définit une méthode `ProvidesFormatInfo` qui utilise des instructions is pour déterminer si un objet fournit une implémentation de IFormatProvider et teste si l'objet est `null`. Il utilise également le modèle d'élément ignoré pour gérer les objets non null de n'importe quel autre type.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      object[] objects = { CultureInfo.CurrentCulture,
                           CultureInfo.CurrentCulture.DateTimeFormat,
                           CultureInfo.CurrentCulture.NumberFormat,
                           new ArgumentException(), null };
      foreach (var obj in objects)
         ProvidesFormatInfo(obj);
   }

   private static void ProvidesFormatInfo(object obj)
   {
      switch (obj)
      {
         case IFormatProvider fmt:
            Console.WriteLine($"{fmt} object");
            break;
         case null:
            Console.Write("A null object reference: ");
            Console.WriteLine("Its use could result in a NullReferenceException");
            break;
         case object _:
            Console.WriteLine("Some object type without format information");
            break;
      }
   }
}
// The example displays the following output:
//    en-US object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
//    Some object type without format information
//    A null object reference: Its use could result in a NullReferenceException
```

Appels à des méthodes avec des paramètres out

Quand vous appelez la méthode `Deconstruct` pour déconstruire un type défini par l'utilisateur (une instance d'une classe, d'une structure ou d'une interface), vous pouvez ignorer les valeurs d'arguments `out` individuels. Vous pouvez cependant aussi ignorer la valeur d'arguments `out` lors de l'appel de n'importe quelle méthode avec un paramètre out.

L'exemple suivant appelle la méthode DateTime.TryParse (String, out DateTime) pour déterminer si la représentation sous forme de chaîne d'une date est valide dans la culture actuelle. Comme l'exemple concerne ici uniquement la validation de la chaîne de date et pas son analyse pour extraire la date, l'argument `out` de la méthode est un élément ignoré.

```
using System;

public class Example
{
   public static void Main()
   {
      string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                              "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                              "5/01/2018 14:57:32.80 -07:00",
                              "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                              "Fri, 15 May 2018 20:10:57 GMT" };
      foreach (string dateString in dateStrings)
      {
         if (DateTime.TryParse(dateString, out _))
            Console.WriteLine($"'{dateString}': valid");
         else
            Console.WriteLine($"'{dateString}': invalid");
      }
   }
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

Élément ignoré autonome

Vous pouvez utiliser un élément ignoré autonome pour indiquer une variable que vous choisissez d'ignorer. L'exemple suivant utilise un élément ignoré autonome pour ignorer l'objet Task retourné par une opération asynchrone. Ceci a pour effet de supprimer l'exception levée par l'opération au moment où elle se termine.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static async Task Main(string[] args)
   {
      await ExecuteAsyncMethods();
   }

   private static async Task ExecuteAsyncMethods()
   {
      Console.WriteLine("About to launch a task...");
      _ = Task.Run(() => { var iterations = 0;
                           for (int ctr = 0; ctr < int.MaxValue; ctr++)
                              iterations++;
                           Console.WriteLine("Completed looping operation...");
                           throw new InvalidOperationException();
                         });
      await Task.Delay(5000);
      Console.WriteLine("Exiting after 5 second delay");
   }
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Notez que `_` est aussi un identificateur valide. Quand il est utilisé en dehors d'un contexte pris en charge, `_` est traité non pas comme élément ignoré, mais comme variable valide. Si un identificateur nommé `_` est déjà dans l'étendue, l'utilisation de `_` comme élément ignoré autonome peut provoquer :

- Une modification accidentelle de la valeur de la variable `_` dans l'étendue en lui affectant la valeur de l'élément ignoré prévu. Exemple :

```
private static void ShowValue(int _)
{
   byte[] arr = { 0, 0, 1, 2 };
   _ = BitConverter.ToInt32(arr, 0);
   Console.WriteLine(_);
}
// The example displays the following output:
//       33619968
```

- Une erreur de compilateur pour violation de sécurité du type. Exemple :

```
private static bool RoundTrips(int _)
{
   string value = _.ToString();
   int newValue = 0;
   _ = Int32.TryParse(value, out newValue);
   return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Erreur du compilateur CS0136 : « Impossible de déclarer une variable locale ou un paramètre nommé '_' dans cette portée, car ce nom est utilisé dans une portée locale englobante pour définir une variable locale ou un paramètre. » Par exemple :

```
public void DoSomething(int _)
{
 var _ = GetValue(); // Error: cannot declare local _ when one is already in scope
}
// The example displays the following compiler error:
// error CS0136:
//       A local or parameter named '_' cannot be declared in this scope
//       because that name is used in an enclosing local scope
//       to define a local or parameter
```

Voir aussi

- Déconstruction de tuples et d'autres types
- `is` mot
- `switch` mot

Génériques (guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Les génériques introduisent le concept de paramètres de type dans .NET, ce qui permet de concevoir des classes et des méthodes qui diffèrent la spécification d'un ou plusieurs types jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code client. Par exemple, en utilisant un paramètre de type générique `T`, vous pouvez écrire une seule classe qui peut être utilisée par un autre code client sans impliquer le coût ou le risque des casts ou des opérations de boxing du runtime, comme illustré ici :

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Les classes et les méthodes génériques combinent la réutilisabilité, la cohérence des types et l'efficacité d'une façon que leurs équivalents non génériques ne peuvent pas. Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux. L' System.Collections.Generic espace de noms contient plusieurs classes de collection génériques. Les collections non génériques, telles que, ne ArrayList sont pas recommandées et sont conservées à des fins de compatibilité. Pour plus d'informations, consultez Génériques en .NET.

Bien sûr, vous pouvez également créer des types et des méthodes génériques personnalisés pour fournir des solutions et des modèles de conception généralisés qui soient efficaces et de type sécurisé. L'exemple de code suivant montre une classe de liste liée générique simple, à des fins de démonstration. (Dans la plupart des cas, vous devez utiliser la List<T> classe fournie par .net au lieu de créer la vôtre.) Le paramètre `T` de type est utilisé dans plusieurs emplacements où un type concret est normalement utilisé pour indiquer le type de l'élément stocké dans la liste. Il est utilisé de la façon suivante :

- Comme le type d'un paramètre de méthode dans la méthode `AddHead`.
- Comme le type de retour de la propriété `Data` de la classe imbriquée `Node`.
- Comme le type de membre privé `data` de la classe imbriquée.

Notez que `T` est disponible pour la classe imbriquée `Node`. Quand `GenericList<T>` est instancié avec un type concret, par exemple comme un `GenericList<int>`, chaque occurrence de `T` est remplacée par `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

L'exemple de code suivant montre comment le code client utilise la classe générique `GenericList<T>` pour créer une liste d'entiers. En changeant l'argument de type, vous pouvez facilement modifier le code suivant pour créer des listes de chaînes ou tout autre type personnalisé :

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Vue d'ensemble des génériques

- Utilisez des types génériques pour optimiser la réutilisation de code, la sécurité des types et les performances.
- L'utilisation la plus courante des génériques est de créer des classes de collection.
- La bibliothèque de classes .NET contient plusieurs classes de collection génériques dans l' System.Collections.Generic espace de noms. Celles-ci doivent être utilisées chaque fois que c'est possible, à la place de classes comme ArrayList dans l'espace de noms System.Collections.
- Vous pouvez créer vos propres interfaces, classes, méthodes, événements et délégués génériques.
- Les classes génériques peuvent être contraintes pour permettre l'accès à des méthodes sur des types de données particuliers.
- Des informations sur les types qui sont utilisés dans un type de données générique peuvent être obtenues à l'exécution à l'aide de la réflexion.

Sections connexes

- Paramètres de type générique
- Contraintes sur les paramètres de type
- Classes génériques
- Interfaces génériques
- Méthodes génériques
- Délégués génériques
- Différences entre les modèles C++ et les génériques C#
- Génériques et réflexion
- Génériques dans le runtime

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Types

- <typeparam>
- <typeparamref>
- Génériques en .NET

Iterators

02/11/2020 • 10 minutes to read • Edit Online

Presque chaque programme que vous écrivez doit itérer au sein d'une collection. Vous allez écrire du code qui examine chaque élément d'une collection.

Vous allez également créer des méthodes d'itérateur qui sont des méthodes qui produisent un itérateur (qui est un objet qui traverse un conteneur, en particulier des listes) pour les éléments de cette classe. Vous pouvez les utiliser pour :

- Effectuer une action sur chaque élément d'une collection.
- Énumérer une collection personnalisée.
- Étendre LINQ ou d'autres bibliothèques.
- Créer un pipeline de données où les données circulent efficacement via des méthodes d'itérateur.

Le langage C# fournit des fonctionnalités pour ces deux scénarios. Cet article présente une vue d'ensemble de ces fonctionnalités.

Ce didacticiel comporte plusieurs étapes. Après chaque étape, vous pourrez exécuter l'application et voir la progression. Vous pouvez également afficher ou télécharger l'exemple complet pour cette rubrique. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Itération avec foreach

L'énumération d'une collection est simple : le mot clé `foreach` énumère une collection en exécutant l'instruction incorporée une fois pour chaque élément de la collection :

```
foreach (var item in collection)
{
   Console.WriteLine(item.ToString());
}
```

C'est tout. Pour itérer au sein du contenu d'une collection, l'instruction `foreach` suffit. L'instruction `foreach` n'est cependant pas magique. Elle s'appuie sur deux interfaces génériques définies dans la bibliothèque .NET Core afin de générer le code nécessaire pour itérer au sein d'une collection : `IEnumerable<T>` et `IEnumerator<T>`. Ce mécanisme est expliqué plus en détail ci-dessous.

Ces deux interfaces ont également des contreparties non génériques : `IEnumerable` et `IEnumerator`. Les versions génériques conviennent mieux pour un code moderne.

Sources d'énumération avec des méthodes d'itérateur

Une autre fonctionnalité intéressante du langage C# vous permet de construire des méthodes qui créent une source pour une énumération. Elles sont appelées *méthodes d'itérateur*. Une méthode d'itérateur définit comment générer les objets dans une séquence sur demande. Vous utilisez les mots clés contextuels `yield return` pour définir une méthode d'itérateur.

Vous pouvez par exemple écrire cette méthode pour produire la séquence d'entiers de 0 à 9 :

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

Le code ci-dessus montre des instructions `yield return` distinctes pour mettre en évidence le fait que vous pouvez utiliser plusieurs instructions `yield return` discrètes dans une méthode d'itérateur. Vous pouvez (et c'est souvent le cas) utiliser d'autres constructions du langage pour simplifier le code d'une méthode d'itérateur. La définition de méthode ci-dessous produit exactement la même séquence de nombres :

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

Vous n'êtes pas obligé de choisir l'une ou l'autre. Vous pouvez avoir autant d'instructions `yield return` que nécessaire pour répondre aux besoins de votre méthode :

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Il s'agit de la syntaxe de base. Prenons un exemple réel où vous devez écrire une méthode d'itérateur. Imaginez que vous êtes sur un projet IoT et que les capteurs d'un appareil génèrent un flux de données très important. Pour obtenir un aperçu des données, vous pouvez écrire une méthode qui échantillonne les éléments de données à un certain intervalle. Cette petite méthode d'itérateur peut effectuer ce travail :

```
public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Il existe une restriction importante sur les méthodes d'itérateur : vous ne pouvez pas avoir à la fois une instruction

`return` et une instruction `yield return` dans la même méthode. Ce qui suit ne peut pas être compilé :

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

Cette restriction n'est normalement pas un problème. Vous avez le choix entre utiliser `yield return` dans toute la méthode ou de diviser la méthode d'origine en plusieurs méthodes, certaines utilisant `return` et d'autres utilisant `yield return`.

Vous pouvez modifier légèrement la dernière méthode pour utiliser `yield return` partout :

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

Parfois, la bonne réponse est de fractionner une méthode d'itérateur en deux méthodes différentes. Une qui utilise `return` et l'autre qui utilise `yield return`. Considérez une situation où vous voulez retourner une collection vide, ou bien les 5 premiers nombres impairs, en vous basant sur un argument booléen. Vous pouvez écrire ceci sous la forme de ces deux méthodes :

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Examinez les méthodes ci-dessus. La première utilise l'instruction `return` standard pour retourner une collection vide ou l'itérateur créé par la deuxième méthode. La deuxième méthode utilise l'instruction `yield return` pour

créer la séquence demandée.

Aller plus loin avec `foreach`

L'instruction `foreach` se développe en un idiome standard qui utilise les interfaces `IEnumerable<T>` et `IEnumerator<T>` pour itérer à travers tous les éléments d'une collection. Elle réduit également les erreurs que font les développeurs en ne gérant pas correctement les ressources.

Le compilateur traduit la boucle `foreach` présentée dans le premier exemple en quelque chose de similaire à cette construction :

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

La construction ci-dessus représente le code généré par le compilateur C# dans la version 5 et ultérieure. Avant la version 5, la variable `item` avait une étendue différente :

```
// C# versions 1 through 4:
IEnumerator<int> enumerator = collection.GetEnumerator();
int item = default(int);
while (enumerator.MoveNext())
{
    item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

Ceci a été changé, car le comportement antérieur pouvait entraîner des bogues subtils et difficiles à diagnostiquer impliquant des expressions lambda. Pour plus d'informations sur les expressions lambda, voir Expressions lambda.

Le code exact généré par le compilateur est un peu plus compliqué et gère les cas où l'objet retourné par `GetEnumerator()` implémente l'interface `IDisposable`. L'expansion complète génère un code qui ressemble davantage à celui-ci :

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    } finally
    {
        // dispose of enumerator.
    }
}
```

La manière dont l'énumérateur est éliminé dépend des caractéristiques du type d'`enumerator`. En règle générale, la clause `finally` se développe en :

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

Cependant, si le type d'`enumerator` est un type sealed et qu'il n'existe pas de conversion implicite du type d'`enumerator` en `IDisposable`, la clause `finally` se développe en un bloc vide :

```
finally
{
}
```

S'il existe une conversion implicite du type d'`enumerator` en `IDisposable` et si `enumerator` est un type valeur qui n'autorise pas les valeurs Null, la clause `finally` se développe en :

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Heureusement, vous n'avez pas besoin de mémoriser tous ces détails. L'instruction `foreach` gère toutes ces subtilités pour vous. Le compilateur génère le code correct pour toutes ces constructions.

Présentation des délégués

18/03/2020 • 4 minutes to read • Edit Online

Les délégués fournissent un mécanisme de *liaison tardive* dans .NET. La liaison tardive signifie que vous créez un algorithme où l'appelant fournit également au moins une méthode qui implémente une partie de l'algorithme.

Par exemple, prenez le tri d'une liste d'étoiles dans une application d'astronomie. Vous pouvez choisir de trier les étoiles d'après leur distance par rapport à la terre, d'après leur magnitude ou d'après leur luminosité perçue.

Dans tous ces cas-là, la méthode Sort() effectue essentiellement la même chose : elle organise les éléments dans la liste en se basant sur une comparaison. Le code qui compare deux étoiles est différent pour chacun des ordres de tri.

Ces genres de solutions sont utilisés dans les logiciels depuis un demi siècle. Le concept de délégué du langage C# fournit une prise en charge linguistique de première classe et la sécurité de type autour du concept.

Comme vous le verrez plus loin dans cette série, le code C# que vous écrivez pour des algorithmes comme celui-ci est de type sécurisé. Il s'appuie sur le langage et le compilateur pour s'assurer que les types correspondent à des arguments et des types de retour.

Objectifs de conception de langage pour les délégués

Les concepteurs de langage ont énuméré plusieurs objectifs pour la fonctionnalité qui est finalement devenue celle des délégués.

L'équipe souhaitait une construction de langage commune pouvant être utilisée pour tout algorithme à liaison tardive. Cela permet aux développeurs d'apprendre un concept et d'utiliser ce même concept pour résoudre de nombreux problèmes logiciels différents.

Ensuite, l'équipe souhaitait prendre en charge les appels de méthodes uniques et multicasts (les délégués multicasts sont des délégués qui chaînent plusieurs appels de méthode. Vous en verrez des exemples plus loin dans cette série).

L'équipe souhaitait que les délégués prennent en charge la même sécurité de type que celle attendue par les développeurs pour toutes les constructions C#.

Pour finir, l'équipe a reconnu qu'un modèle d'événement était un modèle spécifique où les délégués (ou tout algorithme de liaison tardive) étaient très utiles. L'équipe souhaitait s'assurer que le code pour les délégués pouvait fournir la base pour le modèle d'événement .NET.

Le résultat de tout ce travail est la prise en charge des délégués et des événements en C# et .NET. Les autres articles de cette section traitent des fonctionnalités de langage, de la prise en charge de la bibliothèque, et des idiomes courants utilisés quand vous travaillez avec des délégués.

Vous découvrirez le mot clé `delegate` et le code qu'il génère. Vous découvrirez les fonctionnalités de la classe `System.Delegate` et comment elles sont utilisées. Vous découvrirez comment créer des délégués de type sécurisé et comment créer des méthodes qui peuvent être appelées par l'intermédiaire de délégués. Vous découvrirez aussi comment utiliser des délégués et des événements à l'aide d'expressions lambda. Vous verrez où les délégués deviennent l'un des blocs de construction de LINQ. Vous découvrirez que les délégués sont la base du modèle d'événement .NET, et comment ils diffèrent.

Globalement, vous verrez comment les délégués forment une partie intégrante de la programmation dans .NET et de l'utilisation des API du framework.

Nous pouvons commencer.

Suivant

System.Delegate et le mot clé `delegate`

02/11/2020 • 12 minutes to read • Edit Online

Précédent

Cet article décrit les classes dans .NET qui prennent en charge les délégués et comment elles sont mappées au `delegate` mot clé.

Définir les types délégués

Commençons par le mot clé 'delegate', car c'est l'élément principal que vous utilisez quand vous travaillez avec des délégués. Le code que le compilateur génère quand vous utilisez le mot clé `delegate` mappe aux appels de méthode qui appellent des membres des classes Delegate et MulticastDelegate.

Vous définissez un type délégué à l'aide d'une syntaxe similaire à la définition d'une signature de méthode. Vous venez d'ajouter le mot clé `delegate` à la définition.

Continuons à utiliser la méthode List.Sort() comme dans notre exemple. La première étape consiste à créer un type pour le délégué de comparaison :

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

Le compilateur génère une classe, dérivée de `System.Delegate`, qui correspond à la signature utilisée (dans le cas présent, une méthode qui retourne un entier et qui a deux arguments). Le type de ce délégué est `Comparison`. Le type délégué `Comparison` est un type générique. Pour plus d'informations sur les génériques, cliquez ici.

Notez que la syntaxe peut sembler déclarer une variable, alors qu'elle déclare en fait un *type*. Vous pouvez définir des types délégués dans des classes, directement dans des espaces de noms, ou même dans l'espace de noms global.

> **NOTE**
> La déclaration de types délégués (ou d'autres types) directement dans l'espace de noms global n'est pas recommandée.

Le compilateur génère également des gestionnaires d'ajout et de suppression pour ce nouveau type, afin que les clients de cette classe puissent ajouter et supprimer des méthodes dans la liste d'invocation d'une instance. Le compilateur exige que la signature de la méthode ajoutée ou supprimée corresponde à la signature utilisée lors de la déclaration de la méthode.

Déclarer des instances de délégués

Après avoir défini le délégué, vous pouvez créer une instance de ce type. Comme pour toutes les variables en C#, vous ne pouvez pas déclarer d'instances de délégué directement dans un espace de noms, ni dans l'espace de noms global.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

Le type de la variable est le type délégué défini précédemment, `Comparison<T>`. Le nom de la variable est `comparator`.

Cet extrait de code ci-dessus a déclaré une variable membre à l'intérieur d'une classe. Vous pouvez également déclarer des variables de délégués qui sont des variables locales, ou bien des arguments de méthodes.

Appeler des délégués

Vous appelez les méthodes qui se trouvent dans la liste d'invocation d'un délégué en appelant ce dernier. À l'intérieur de la méthode `Sort()`, le code appelle la méthode de comparaison pour déterminer l'ordre dans lequel placer les objets :

```
int result = comparator(left, right);
```

Dans la ligne ci-dessus, le code *appelle* la méthode attachée au délégué. Vous traitez la variable comme un nom de méthode et vous l'appelez à l'aide de la syntaxe d'appel de méthode normale.

Cette ligne de code effectue une hypothèse hasardeuse : il n'existe aucune garantie qu'une cible a été ajoutée au délégué. Si aucune cible n'a été attachée, la ligne ci-dessus entraîne la levée de `NullReferenceException`. Les idiomes utilisés pour résoudre ce problème sont plus compliqués qu'un simple contrôle de valeur Null. Ils sont traités plus loin dans cette série.

Assigner, ajouter et supprimer des cibles d'appel

Voyons comment un type délégué est défini et comment les instances de délégué sont déclarées et appelées.

Les développeurs qui souhaitent utiliser la méthode `List.Sort()` doivent définir une méthode dont la signature correspond à la définition du type délégué, puis l'assigner au délégué utilisé par la méthode sort. Cette assignation ajoute la méthode à la liste d'invocation de cet objet délégué.

Supposons que vous souhaitiez trier une liste de chaînes en fonction de leur longueur. Votre fonction de comparaison peut être la suivante :

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

La méthode est déclarée en tant que méthode privée. C'est parfait. Vous ne voulez peut-être pas que cette méthode fasse partie de votre interface publique. Elle peut toujours être utilisée comme méthode de comparaison quand elle est attachée à un délégué. Cette méthode est attachée à la liste cible de l'objet délégué dans le code appelant, qui peut y accéder par l'intermédiaire de ce délégué.

Vous créez cette relation en passant cette méthode à la méthode `List.Sort()` :

```
phrases.Sort(CompareLength);
```

Notez que le nom de la méthode est utilisé sans parenthèses. L'utilisation de la méthode comme argument indique au compilateur de convertir la référence de méthode en une référence pouvant être utilisée comme cible d'invocation de délégué, puis d'attacher cette méthode comme cible d'invocation.

Vous pouvez également avoir déclaré explicitement une variable de type `Comparison<string>` et effectué une assignation :

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Quand la méthode utilisée comme cible du délégué est petite, il est courant d'utiliser la syntaxe des expressions lambda pour effectuer l'assignation :

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

L'utilisation d'expressions lambda pour les cibles de délégué est traitée plus loin dans une section ultérieure.

L'exemple Sort() attache généralement une méthode cible unique au délégué. Toutefois, les objets délégués prennent en charge les listes d'invocation comprenant plusieurs méthodes cibles attachées à un objet délégué.

Classes Delegate et MulticastDelegate

La prise en charge du langage décrite ci-dessus fournit les fonctionnalités et la prise en charge généralement nécessaires pour utiliser des délégués. Ces fonctionnalités sont basées sur deux classes du framework .NET Core : Delegate et MulticastDelegate.

La `System.Delegate` classe et sa sous-classe directe unique, `System.MulticastDelegate` fournissent la prise en charge de l'infrastructure pour la création de délégués, l'inscription de méthodes en tant que cibles de délégué et l'appel de toutes les méthodes inscrites en tant que cible de délégué.

Il est intéressant de noter que les classes `System.Delegate` et `System.MulticastDelegate` ne sont pas elles-mêmes des types délégués. Elles servent de base à tous les types délégués spécifiques. Ce même processus de conception du langage a stipulé que vous ne pouvez pas déclarer une classe qui dérive de `Delegate` ou de `MulticastDelegate`. Les règles du langage C# l'interdisent.

À la place, le compilateur C# crée des instances d'une classe dérivée de `MulticastDelegate` quand vous utilisez le mot clé du langage C# pour déclarer des types délégués.

Cette conception trouve ses racines dans la première version Release de C# et du .NET. L'équipe de conception avait pour objectif de s'assurer que le langage appliquait la cohérence des types lors de l'utilisation de délégués. Il était important de garantir que les délégués étaient appelés avec le type et le nombre d'arguments appropriés. Et que tout type de retour était correctement indiqué au moment de la compilation. Les délégués faisaient partie de la version Release du .NET 1.0, qui était antérieure aux génériques.

La création, par le compilateur, des classes déléguées concrètes qui représentaient la signature de méthode utilisée constituait la meilleure façon d'appliquer la cohérence des types.

Même si vous ne pouvez pas créer des classes dérivées directement, vous utiliserez les méthodes définies sur ces classes. Examinons les méthodes les plus courantes que vous utiliserez quand vous travaillerez avec des délégués.

Le premier et le plus important point à retenir est que chaque délégué avec lequel vous travaillez est dérivé de `MulticastDelegate`. Un délégué multicast signifie que plusieurs méthodes cibles peuvent être appelées lors d'un appel par l'intermédiaire d'un délégué. La conception d'origine envisageait de distinguer les délégués où une seule méthode cible pouvait être attachée et appelée, et les délégués où plusieurs méthodes cibles pouvaient être attachées et appelées. En réalité, cette distinction s'est avérée moins utile que prévu. Les deux différentes classes étaient déjà créées et sont présentes dans le framework depuis la première version Release publique de celui-ci.

Les méthodes que vous utiliserez le plus avec les délégués sont `Invoke()` et `BeginInvoke()` / `EndInvoke()`.

`Invoke()` appelle toutes les méthodes qui ont été attachées à une instance de délégué particulière. Comme vous l'avez vu ci-dessus, vous appelez généralement des délégués à l'aide de la syntaxe d'appel de méthode sur la variable de délégué. Comme vous le verrez plus loin dans cette série, il existe des modèles qui fonctionnent directement avec ces méthodes.

Maintenant que vous avez vu la syntaxe du langage et les classes qui prennent en charge les délégués, examinons la façon dont les délégués fortement typés sont utilisés, créés et appelés.

Next

Délégués fortement typés

18/03/2020 • 5 minutes to read • Edit Online

Précédent

Dans l'article précédent, vous avez vu que vous pouviez créer des types délégués spécifiques à l'aide du mot clé `delegate`.

La classe abstraite Delegate fournit l'infrastructure pour l'invocation et le couplage faible. Les types délégués concrets deviennent beaucoup plus utiles en adoptant et en appliquant la sécurité de type pour les méthodes qui sont ajoutées à la liste d'invocation d'un objet délégué. Quand vous utilisez le mot clé `delegate` et que vous définissez un type délégué concret, le compilateur génère ces méthodes.

Dans la pratique, cela conduirait à la création de types délégués chaque fois que vous avez besoin d'une signature de méthode différente. Ce travail peut devenir fastidieux au bout d'un moment. Chaque nouvelle fonctionnalité nécessite de nouveaux types délégués.

Heureusement, cela n'est pas nécessaire. Le framework .NET Core contient plusieurs types que vous pouvez réutiliser chaque fois que vous avez besoin de types délégués. Il s'agit de définitions génériques. Vous pouvez ainsi déclarer des personnalisations quand vous avez besoin de nouvelles déclarations de méthode.

Le premier de ces types est le type Action et plusieurs variantes :

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

Le modificateur `in` sur l'argument de type générique est traité dans l'article sur la covariance.

Il existe des variantes du délégué `Action` qui contiennent jusqu'à 16 arguments, comme Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. Il est important que ces définitions utilisent différents arguments génériques pour chacun des arguments de délégués : cela offre une flexibilité maximale. Les arguments de méthode ne doivent pas obligatoirement être du même type, mais il peuvent l'être.

Utilisez l'un des types `Action` pour tout type délégué ayant un type de retour void.

Le framework inclut également plusieurs types délégués génériques que vous pouvez utiliser pour les types délégués qui retournent des valeurs :

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

Le modificateur `out` sur l'argument de type générique de résultat est traité dans l'article sur la covariance.

Il existe des variantes du délégué `Func` qui contiennent jusqu'à 16 arguments d'entrée, comme Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. Par convention, le type du résultat est toujours le dernier paramètre de type dans toutes les déclarations `Func`.

Utilisez l'un des types `Func` pour tout type délégué qui retourne une valeur.

Il y a aussi unePredicate<T> type pour un délégué qui retourne un test sur une seule valeur :

```
public delegate bool Predicate<in T>(T obj);
```

Vous remarquerez que pour tout type `Predicate`, il existe un type `Func` structurellement équivalent, par exemple :

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Ces deux types peuvent vous sembler équivalents, mais ils ne le sont pas. Ces deux variables ne sont pas interchangeables. Une variable d'un type ne peut pas être assignée à l'autre type. Le système de type C# utilise les noms des types définis, pas la structure.

Toutes ces définitions de types délégués dans la bibliothèque .NET Core doivent signifier que vous n'avez pas besoin de définir un nouveau type délégué pour toute nouvelle fonctionnalité que vous créez et qui nécessite des délégués. Ces définitions génériques doivent fournir tous les types délégués dont vous avez besoin dans la plupart des situations. Il vous suffit d'instancier l'un de ces types avec les paramètres de type nécessaires. Dans le cas des algorithmes qui peuvent être rendus génériques, ces délégués peuvent être utilisés en tant que types génériques.

Cela devrait vous procurer un gain de temps et réduire le nombre de nouveaux types que vous devez créer pour travailler avec des délégués.

Dans l'article suivant, vous verrez plusieurs schémas d'utilisation courants des délégués.

Suivant

Modèles courants pour les délégués

02/11/2020 • 16 minutes to read • Edit Online

Précédent

Les délégués fournissent un mécanisme qui autorise des conceptions logicielles impliquant un couplage minimal entre les composants.

LINQ constitue un excellent exemple de ce genre de conception. Le modèle d'expression de requête LINQ s'appuie sur des délégués pour toutes ses fonctionnalités. Prenons l'exemple simple suivant :

```
var smallNumbers = numbers.Where(n => n < 10);
```

Il filtre la séquence de nombres pour ne sélectionner que ceux qui sont inférieurs à 10. La méthode `Where` utilise un délégué qui détermine les éléments d'une séquence qui correspondent au filtre. Quand vous créez une requête LINQ, vous fournissez l'implémentation du délégué à cette fin spécifique.

Le prototype de la méthode Where est le suivant :

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool>
predicate);
```

Cet exemple est répété avec toutes les méthodes qui font partie de LINQ. Elles s'appuient toutes sur des délégués pour le code qui gère la requête spécifique. Ce modèle de conception d'API est très puissant, une fois appris et compris.

Cet exemple simple montre que les délégués nécessitent très peu de couplage entre les composants. Vous n'avez pas besoin de créer de classe qui dérive d'une classe de base particulière. Vous n'avez pas besoin d'implémenter une interface spécifique. La seule exigence consiste à fournir l'implémentation d'une méthode qui est essentielle à la tâche en question.

Création de vos propres composants avec des délégués

Étendons cet exemple en créant un composant à l'aide d'une conception qui s'appuie sur des délégués.

Nous allons définir un composant qui peut être utilisé pour des messages de journaux dans un système de grande taille. Les composants de bibliothèque pourraient être utilisés dans de nombreux environnements différents, sur plusieurs plateformes différentes. Il existe de nombreuses fonctionnalités courantes dans le composant qui gère les journaux. Il doit accepter des messages en provenance de tout composant du système. Ces messages auront des priorités différentes, que le composant principal peut gérer. Les messages doivent avoir des horodatages dans leur forme finale archivée. Pour des scénarios avancés, vous pouvez filtrer les messages en fonction du composant source.

Il y a un aspect de la fonctionnalité qui changera souvent : l'emplacement d'écriture des messages. Dans certains environnements, ils pourront être écrits dans la console des erreurs. Dans d'autres, il pourra s'agir d'un fichier. Les autres possibilités incluent le stockage de base de données, les journaux des événements du système d'exploitation ou autre stockage de documents.

Il existe également des combinaisons de sortie qui peuvent être utilisées dans différents scénarios. Vous souhaiterez peut-être écrire des messages dans la console et dans un fichier.

Une conception basée sur des délégués offre une très grande souplesse et facilite la prise en charge des

mécanismes de stockage qui pourront être ajoutés à l'avenir.

Dans cette conception, le composant de journal principal peut être une classe non virtuelle, voire sealed. Vous pouvez incorporer n'importe quel jeu de délégués pour écrire les messages sur différents supports de stockage. La prise en charge intégrée des délégués multicast facilite la prise en charge des scénarios où les messages doivent être écrits à plusieurs emplacements (dans un fichier et une console).

Une première implémentation

Commençons par quelque chose de simple : l'implémentation initiale acceptera les nouveaux messages et les écrira à l'aide de n'importe quel délégué attaché. Vous pouvez démarrer avec un délégué qui écrit des messages dans la console.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

La classe statique ci-dessus est la chose la plus simple qui peut fonctionner. Nous devons écrire l'implémentation unique pour la méthode qui écrit des messages dans la console :

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Pour finir, nous devons raccorder le délégué en l'attachant au délégué WriteMessage déclaré dans l'enregistreur d'événements :

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Méthodes

Jusqu'à présent, notre exemple est assez simple, mais il illustre quand même certaines instructions importantes pour les conceptions impliquant des délégués.

L'utilisation des types délégués définis dans le Core Framework permet aux utilisateurs de travailler facilement avec les délégués. Vous n'avez pas besoin de définir de nouveaux types, et les développeurs qui utilisent votre bibliothèque n'ont pas besoin d'apprendre de nouveaux types délégués spécialisés.

Les interfaces utilisées sont le plus minimal et flexible possible : pour créer un enregistreur d'événements de sortie, vous devez créer une méthode. Il peut s'agir d'une méthode statique ou d'une méthode d'instance. Elle peut avoir n'importe quel accès.

Mise en forme de la sortie

Nous allons maintenant rendre cette première version un peu plus robuste et commencer à créer d'autres

mécanismes de journalisation.

Ensuite, nous ajouterons quelques arguments à la méthode `LogMessage()` pour que notre classe de journal crée des messages plus structurés :

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Maintenant, utilisons cet argument `Severity` pour filtrer les messages envoyés à la sortie du journal.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Méthodes

Nous avons ajouté de nouvelles fonctionnalités à l'infrastructure de journalisation. Le composant enregistreur d'événements étant très faiblement couplé à tout mécanisme de sortie, ces nouvelles fonctionnalités peuvent être ajoutées sans aucun impact sur le code qui implémente le délégué enregistreur d'événements.

Au fil du temps, vous verrez d'autres d'exemples montrant comment ce couplage faible autorise une plus grande souplesse dans la mise à jour des parties du site sans aucune modification des autres emplacements. En fait, dans une application plus grande, les classes de sortie de l'enregistreur d'événements peuvent être dans un autre assembly et ne même pas avoir besoin d'être reconstruites.

Création d'un deuxième moteur de sortie

Le composant journal commence à prendre forme. Ajoutons un autre moteur de sortie qui enregistre les

messages dans un fichier. Ce moteur de sortie est légèrement plus complexe. Il s'agit d'une classe qui encapsule les opérations de fichiers et garantit que le fichier est toujours fermé après chaque écriture. Cela permet de s'assurer que toutes les données sont vidées sur le disque après chaque génération de message.

Voici cet enregistreur d'événements basé sur fichier :

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Une fois que nous avons créé cette classe, nous pouvons l'instancier, et elle attache sa méthode LogMessage au composant enregistreur d'événements :

```
var file = new FileLogger("log.txt");
```

Ces deux méthodes ne s'excluent pas mutuellement. Nous pourrions attacher les deux méthodes de journalisation et générer des messages dans la console et dans un fichier :

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the static class we utilized
earlier
```

Plus tard, même dans la même application, nous pourrions supprimer l'un des délégués sans provoquer d'autre problème dans le système :

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Méthodes

Nous avons maintenant ajouté un deuxième gestionnaire de sortie pour le sous-système de journalisation. Celui-ci a besoin d'un peu plus d'infrastructure pour correctement prendre en charge le système de fichiers. Le délégué est une méthode d'instance. Il s'agit aussi d'une méthode privée. Une plus grande accessibilité n'est pas nécessaire, car l'infrastructure de délégué peut connecter les délégués.

Ensuite, la conception basée sur les délégués autorise plusieurs méthodes de sortie sans code supplémentaire. Vous n'avez pas besoin de créer d'infrastructure supplémentaire pour prendre en charge plusieurs méthodes de sortie. Elles viennent simplement s'ajouter comme autre méthode dans la liste d'invocation.

Faites particulièrement attention au code de la méthode de sortie de journalisation de fichier. Cette méthode est codée pour s'assurer qu'elle ne lève pas d'exception. Même si ce n'est pas toujours strictement nécessaire, cette pratique est souvent recommandée. Si l'une des méthodes délégué lève une exception, les délégués restants qui sont sur la liste d'invocation ne sont pas appelés.

Dernière remarque : l'enregistreur d'événements de fichier doit gérer ses ressources en ouvrant et en fermant le fichier à chaque message de journal. Vous pouvez choisir de laisser le fichier ouvert et d'implémenter IDisposable pour fermer le fichier une fois que vous avez terminé. Les deux méthodes ont leurs avantages et leurs inconvénients. Elles créent toutes les deux un peu plus de couplage entre les classes.

Pour prendre en charge ces deux scénarios, il est inutile de mettre à jour le code de la classe Logger.

Gestion des délégués Null

Pour finir, nous allons mettre à jour la méthode LogMessage afin qu'elle soit robuste pour les cas où aucun mécanisme de sortie n'est sélectionné. L'implémentation actuelle lève une `NullReferenceException` quand aucune liste d'invocation n'est attachée au délégué `WriteMessage`. Vous préférerez peut-être une conception qui continue en mode silencieux quand aucune méthode n'a été attachée. Pour cela, vous pouvez utiliser l'opérateur conditionnel null, combiné avec la méthode `Delegate.Invoke()` :

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

L'opérateur conditionnel null (`?.`) court-circuite quand l'opérande gauche (ici, `WriteMessage`) est null, ce qui signifie que le système ne tente pas de consigner un message dans le journal.

Vous ne trouverez pas la méthode `Invoke()` répertoriée dans la documentation de `System.Delegate` ou `System.MulticastDelegate`. Le compilateur génère une méthode `Invoke` de type sécurisé pour tout type de délégué déclaré. Dans cet exemple, cela signifie que `Invoke` prend un seul argument `string` et a un type de retour void.

Récapitulatif des méthodes

Nous venons de voir le début d'un composant de journal qui peut être étendu avec d'autres enregistreurs, ainsi que d'autres fonctionnalités. Grâce à l'utilisation de délégués dans la conception, ces différents composants sont très faiblement couplés. Cela présente plusieurs avantages. Il est très facile de créer de nouveaux mécanismes de sortie et de les attacher au système. Ces autres mécanismes n'ont besoin que d'une seule méthode : celle qui écrit le message du journal. Cette conception offre une grande souplesse lors de l'ajout de nouvelles fonctionnalités. Le contrat requis pour tout enregistreur consiste à implémenter une méthode. Il peut s'agir d'une méthode d'instance ou statique. Elle peut être publique, privée ou avoir tout autre accès légal.

La classe Logger peut apporter des améliorations ou des modifications sans introduire de modifications avec rupture. Comme toute classe, vous ne pouvez pas modifier l'API publique sans risque de modifications avec rupture. Toutefois, le couplage entre l'enregistreur d'événements et les moteurs de sortie étant uniquement par

l'intermédiaire du délégué, aucun autre type (tel que les interfaces ou classes de base) n'est appelé. Le couplage est aussi réduit que possible.

Next

Introduction aux événements

02/11/2020 • 6 minutes to read • Edit Online

Précédent

Les événements sont, comme les délégués, un mécanisme de *liaison tardive*. En fait, les événements reposent sur la prise en charge linguistique des délégués.

Ils permettent à un objet de signaler (à tous les composants du système intéressés), que quelque chose s'est produit. Tous les autres composants peuvent s'abonner à l'événement et être averti quand un événement est déclenché.

Vous avez sans doute déjà utilisé des événements durant vos tâches de programmation. De nombreux systèmes graphiques ont un modèle d'événement pour signaler l'interaction utilisateur. Ces événements signalent les mouvements de la souris, les pressions sur des boutons et autres interactions similaires. C'est l'un des scénarios les plus courants, mais certainement pas le seul, où des événements sont utilisés.

Vous pouvez définir les événements qui doivent être déclenchés pour vos classes. Quand vous travaillez avec des événements, vous devez prendre en compte le fait qu'il se peut qu'aucun objet ne soit inscrit pour un événement particulier. Vous devez écrire votre code pour qu'il ne déclenche pas d'événement quand aucun détecteur n'est configuré.

L'abonnement à un événement crée également un couplage entre deux objets (la source de l'événement et le récepteur d'événements). Vous devez vérifier que le récepteur d'événements annule l'abonnement à la source de l'événement quand il n'est plus intéressé par les événements.

Objectifs de conception pour la prise en charge des événements

La conception de langage pour les événements cible ces objectifs :

- Activez un couplage très minimal entre une source d'événement et un récepteur d'événements. Ces deux composants peuvent ne pas avoir été écrits par la même organisation, et peuvent même être mis à jour d'après une planification totalement différente.
- Il doit être très simple de s'abonner à un événement et de se désabonner de ce même événement.
- Les sources d'événements doivent prendre en charge plusieurs abonnés aux événements. Elles doivent aussi prendre en charge les scénarios où aucun abonné aux événements n'est attaché.

Vous pouvez constater que les objectifs pour les événements sont très similaires aux objectifs pour les délégués. C'est pourquoi la prise en charge linguistique des événements repose sur la prise en charge linguistique des délégués.

Prise en charge linguistique pour les événements

La syntaxe de définition des événements, et d'abonnement ou d'annulation des abonnements, est une extension de la syntaxe des délégués.

Pour définir un événement, vous devez utiliser le mot clé `event` :

```
public event EventHandler<FileListArgs> Progress;
```

Le type de l'événement (`EventHandler<FileListArgs>` dans cet exemple) doit être un type délégué. Vous devez

respecter plusieurs conventions lors de la déclaration d'un événement. En général, le type délégué d'événement a un retour void. Les déclarations d'événements doivent être un verbe ou une phrase verbale. Utilisez des dizaines lorsque l'événement signale une action qui s'est produite. Utilisez un verbe au présent (par exemple, `Closing`) pour signaler quelque chose qui est sur le point de se produire. Souvent, le présent indique que votre classe prend en charge un type de comportement de personnalisation. L'un des scénarios les plus courants consiste à prendre en charge l'annulation. Par exemple, un événement `Closing` peut inclure un argument qui indique si l'opération de fermeture doit continuer ou non. D'autres scénarios peuvent permettre aux appelants de modifier le comportement en mettant à jour les propriétés des arguments de l'événement. Vous pouvez déclencher un événement pour indiquer une action suivante qu'un algorithme effectuera. Le gestionnaire d'événements peut imposer une action différente en modifiant les propriétés de l'argument d'événement.

Quand vous souhaitez déclencher l'événement, vous appelez les gestionnaires d'événements à l'aide de la syntaxe d'appel de délégué :

```
Progress?.Invoke(this, new FileListArgs(file));
```

Comme indiqué dans la section sur les délégués, l'opérateur ?. permet de s'assurer facilement que vous n'essayez pas de déclencher l'événement quand il n'y a aucun abonné à cet événement.

Vous vous abonnez à un événement à l'aide de l'opérateur `+=` :

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

La méthode de gestionnaire a généralement le préfixe « on » suivi du nom de l'événement, comme indiqué ci-dessus.

Vous pouvez vous désinscrire à l'aide de l'opérateur `-=` :

```
fileLister.Progress -= onProgress;
```

Il est important de déclarer une variable locale pour l'expression qui représente le gestionnaire d'événements. Cela garantit que le désabonnement supprime le gestionnaire. Si au lieu de cela vous utilisez le corps de l'expression lambda, vous essayez de supprimer un gestionnaire qui n'a jamais été attaché, ce qui ne fait rien.

Dans l'article suivant, vous en apprendrez davantage sur les modèles d'événements les plus courants et sur les différentes variantes de cet exemple.

Next

Modèles d'événement .NET standard

18/03/2020 • 15 minutes to read • Edit Online

Précédent

Les événements .NET respectent en général quelques modèles connus. La normalisation d'après ces modèles signifie que les développeurs peuvent exploiter les connaissances de ces modèles standard, qui peuvent être appliqués à n'importe quel programme d'événement .NET.

Un examen de ces modèles standard vous permettra d'acquérir toutes les connaissances nécessaires pour créer des sources d'événements standard, et pour traiter et vous abonner à des événements standard dans votre code.

Signatures de délégués d'événements

La signature standard d'un délégué d'événement .NET est la suivante :

```
void OnEventRaised(object sender, EventArgs args);
```

Le type de retour est void. Les événements sont basés sur des délégués et il s'agit de délégués multicast. Cela permet de prendre en charge plusieurs abonnés pour toute source d'événement. La valeur de retour unique d'une méthode ne s'étend pas à plusieurs abonnés aux événements. Quelle valeur de retour la source de l'événement voit-elle après le déclenchement d'un événement ? Plus loin dans cet article, vous verrez comment créer des protocoles d'événements qui prennent en charge les abonnés aux événements qui signalent des informations à la source de l'événement.

La liste d'arguments contient deux arguments : l'expéditeur et les arguments de l'événement. Le type au moment de compilation de `sender` est `System.Object`, même si vous connaissez probablement un type plus dérivé qui serait toujours correct. Par convention, utilisez `object`.

Le deuxième argument était traditionnellement un type dérivé de `System.EventArgs`. (Vous verrez dans la section suivante que cette convention n'est plus appliquée.) Si votre type d'événement n'a pas besoin d'arguments supplémentaires, vous fournirez toujours les deux arguments. Il existe une valeur spéciale, `EventArgs.Empty`, que vous devez utiliser pour indiquer que votre événement ne contient pas d'informations supplémentaires.

Commençons par créer une classe qui répertorie les fichiers contenus dans un répertoire ou dans l'un de ses sous-répertoires qui suivent un modèle. Ce composant déclenche un événement pour chaque fichier détecté qui correspond au modèle.

L'utilisation d'un modèle d'événement offre certains avantages en matière de conception. Vous pouvez créer plusieurs détecteurs d'événements qui effectuent des actions différentes quand un fichier recherché est trouvé. La combinaison des différents détecteurs permet de créer des algorithmes plus robustes.

Voici la déclaration d'argument d'événement initiale pour trouver un fichier recherché :

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Bien que ce type ressemble à un petit type « données uniquement », vous devez suivre la convention et faire de lui un type référence (`class`). Cela signifie que l'objet d'argument sera passé par référence, et que les mises à jour des données seront visibles par tous les abonnés. La première version est un objet immuable. Il vaut mieux rendre immuables les propriétés dans votre argument d'événement. Ainsi, un abonné ne peut pas changer les valeurs avant qu'un autre abonné ne les voit. (Il existe des exceptions, comme vous le verrez ci-dessous.)

Ensuite, nous devons créer la déclaration d'événement dans la classe FileSearcher. L'utilisation du type `EventHandler<T>` nous évite de devoir créer une autre définition de type. Il suffit d'utiliser une spécialisation générique.

Nous allons remplir la classe FileSearcher pour rechercher les fichiers qui correspondent à un modèle et déclencher l'événement approprié quand une correspondance est détectée.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Définition et déclenchement d'événements de type champ

Pour ajouter un événement à votre classe, le plus simple consiste à déclarer cet événement en tant que champ public, comme dans l'exemple précédent :

```
public event EventHandler<FileFoundArgs> FileFound;
```

Ce code semble déclarer un champ public, ce qui semble être une mauvaise pratique orientée objet. Vous devez protéger l'accès aux données par l'intermédiaire des propriétés ou méthodes. Bien que cela semble être une mauvaise pratique, le code généré par le compilateur crée en fait des wrappers pour que les objets d'événements soient accessibles uniquement de manière sécurisée. Les seules opérations disponibles sur un événement de type champ sont l'ajout de gestionnaire :

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

et la suppression de gestionnaire :

```
fileLister.FileFound -= onFileFound;
```

Notez qu'il existe une variable locale pour le gestionnaire. Si vous utilisiez le corps de l'expression lambda, la suppression ne fonctionnerait pas correctement. Il s'agirait d'une autre instance du délégué, et l'opération ne ferait rien en mode silencieux.

Le code en dehors de la classe ne peut pas déclencher l'événement et ne peut pas non plus effectuer d'autres opérations.

Retour de valeurs à partir des abonnés aux événements

Notre version simple fonctionne correctement. Ajoutons maintenant une autre fonctionnalité : l'annulation.

Quand vous déclenchez l'événement détecté, les détecteurs doivent pouvoir arrêter le traitement si ce fichier est le dernier recherché.

Les gestionnaires d'événements ne retournant pas de valeur, vous devez communiquer cela par un autre moyen. Le modèle d'événement standard utilise l'objet EventArgs pour inclure des champs que les abonnés aux événements peuvent utiliser pour signaler une annulation.

Deux modèles différents peuvent être utilisés, en fonction de la sémantique du contrat d'annulation. Dans les deux cas, vous ajouterez un champ booléen aux arguments d'événements pour l'événement de fichier trouvé.

L'un des modèles autorise n'importe quel abonné à annuler l'opération. Pour ce modèle, le nouveau champ est initialisé avec la valeur `false`. Tout abonné peut le changer et lui affecter la valeur `true`. Une fois que tous les abonnés ont vu l'événement déclenché, le composant FileSearcher examine la valeur booléenne et effectue une action.

Le deuxième modèle annule l'opération uniquement si tous les abonnés souhaitent l'annuler. Dans ce modèle, le nouveau champ est initialisé pour indiquer que l'opération doit être annulée, et n'importe quel abonné peut le changer pour indiquer que l'opération doit continuer. Une fois que tous les abonnés ont vu l'événement déclenché, le composant FileSearcher examine la valeur booléenne et effectue une action. Il existe une étape supplémentaire dans ce modèle : le composant doit savoir si des abonnés ont vu l'événement. S'il n'y a pas d'abonnés, le champ indique incorrectement une annulation.

Implémentons la première version pour cet exemple. Vous devez ajouter un champ booléen nommé `CancelRequested` au type `FileFoundArgs` :

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set;}

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Ce nouveau champ est automatiquement initialisé avec `false`, la valeur par défaut pour un champ booléen, pour éviter tout risque d'annulation accidentelle. Le seul autre changement à apporter au composant consiste à vérifier l'indicateur après le déclenchement de l'événement, pour voir si l'un des abonnés a demandé une annulation :

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

L'un des avantages de ce modèle est qu'il ne s'agit pas d'une modification avec rupture. Aucun des abonnés n'a demandé d'annulation auparavant, et ils n'en demandent toujours pas. Aucun code d'abonné ne nécessite de mise à jour, sauf s'il souhaite prendre en charge le nouveau protocole d'annulation. Le couplage est très faible.

Nous allons maintenant mettre à jour l'abonné pour qu'il demande une annulation dès qu'il trouve le premier exécutable :

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Ajout d'une autre déclaration d'événement

Nous allons ajouter une autre fonctionnalité et illustrer d'autres idiomes de langage pour les événements. Ajoutons une surcharge de la méthode `Search` qui parcourt tous les sous-répertoires à la recherche de fichiers.

Cette opération pourrait prendre beaucoup de temps dans un répertoire contenant de nombreux sous-répertoires. Ajoutons un événement déclenché au début de chaque nouvelle recherche dans un répertoire. Cela permet aux abonnés de suivre la progression et de tenir l'utilisateur à jour. Tous les exemples que nous avons créés jusqu'à présent sont publics. Faisons de celui-ci un événement interne. Cela signifie que nous pouvons aussi rendre internes les types utilisés pour les arguments.

Nous allons commencer par créer la nouvelle classe dérivée EventArgs pour signaler le nouveau répertoire et la progression.

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Là encore, nous pouvons suivre les recommandations pour créer un type référence immuable pour les arguments d'événements.

Maintenant, définissons l'événement. Cette fois-ci, nous utiliserons une syntaxe différente. En plus d'utiliser la syntaxe du champ, nous pouvons créer explicitement la propriété, avec des gestionnaires d'ajout et de suppression. Dans cet exemple, nous n'aurons pas besoin de code supplémentaire dans ces gestionnaires, mais cet exemple

montre comment les créer.

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;
```

Le code que nous écrivons ici reflète en grande partie le code généré par le compilateur pour les définitions d'événements de champs que nous avons vu précédemment. Nous créons l'événement à l'aide d'une syntaxe très similaire à celle utilisée pour les propriétés. Notez que les gestionnaires ont des noms différents : `add` et `remove`. Il sont appelés pour s'abonner à l'événement ou pour annuler un abonnement. Notez que vous devez également déclarer un champ de stockage privé pour stocker la variable d'événement. Il est initialisé avec la valeur null.

Ensuite, nous allons ajouter la surcharge de la méthode `Search` qui parcourt les sous-répertoires et déclenche les deux événements. Le moyen le plus simple consiste à utiliser un argument par défaut pour indiquer que nous souhaitons rechercher dans tous les répertoires :

```
public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

À ce stade, nous pouvons exécuter l'application qui appelle la surcharge pour rechercher dans tous les sous-répertoires. Il n'existe aucun abonné sur le nouvel événement `ChangeDirectory`, mais l'utilisation de l'idiome `?.Invoke()` garantit que cela fonctionne correctement.

Ajoutons un gestionnaire pour écrire une ligne qui affiche la progression dans la fenêtre de la console.

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};
```

Nous avons vu des modèles qui sont suivis dans tout l'écosystème .NET. En apprenant ces modèles et ces conventions, vous écrirez rapidement du code C# et .NET idiomatique.

Dans le prochain article, nous allons voir quelques changements apportés à ces modèles dans la version la plus récente de .NET.

Suivant

Modèle d'événement .NET Core mis à jour

18/03/2020 • 7 minutes to read • Edit Online

Précédent

L'article précédent décrit les modèles d'événements les plus courants. .NET Core a un modèle plus souple. Dans cette version, la définition `EventHandler<TEventArgs>` n'a plus la contrainte selon laquelle `TEventArgs` doit être une classe dérivée de `System.EventArgs`.

Ceci accroît la flexibilité pour vous et offre une compatibilité descendante. Commençons par la flexibilité. La classe System.EventArgs introduit une méthode : `MemberwiseClone()`, qui crée une copie superficielle de l'objet. Cette méthode doit utiliser la réflexion afin d'implémenter ses fonctionnalités pour n'importe quelle classe dérivée de `EventArgs`. Ces fonctionnalités sont plus faciles à créer dans une classe dérivée spécifique. Cela signifie concrètement que la dérivation depuis System.EventArgs est une contrainte qui limite vos conceptions, mais n'offre aucun avantage supplémentaire. Vous pouvez en fait changer les définitions de `FileFoundArgs` et de `SearchDirectoryArgs` de façon à ce qu'ils ne dérivent pas de `EventArgs`. Le programme fonctionnera exactement de la même façon.

Vous pouvez également changer les `SearchDirectoryArgs` en un struct si vous faites une modification de plus :

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

La modification supplémentaire consiste à appeler le constructeur sans paramètre avant d'entrer dans le constructeur qui initialise tous les champs. Sans cet ajout, les règles de C# signalent que les propriétés sont accessibles avant qu'une valeur leur soit affectée.

Vous ne devez pas changer les `FileFoundArgs` d'une classe (type référence) en un struct (type valeur). La raison en est que le protocole de gestion d'annulation nécessite que les arguments de l'événement soient passés par référence. Si vous aviez apporté cette même modification, la classe de recherche de fichier ne pourrait jamais observer les modifications apportées par les abonnés à l'événement. Une nouvelle copie de la structure serait utilisée pour chaque abonné, et cette copie serait une copie différente de celle vue par l'objet de recherche de fichier.

Voyons ensuite comment cette modification peut avoir une compatibilité descendante. La suppression de la contrainte n'affecte aucun code existant. Les types d'arguments d'événement existants dérivent toujours de `System.EventArgs`. La compatibilité descendante est une des raisons principales pour lesquelles ils continueront à dériver de `System.EventArgs`. Les abonnés à l'événement existants seront les abonnés à un événement qui ont suivi le modèle classique.

Selon une logique similaire, aucun type d'argument d'événement créé maintenant n'aurait aucun abonné dans aucun des codes base existants. Les nouveaux types d'événements qui ne dérivent pas de `System.EventArgs` ne

vont pas arrêter ces codes base.

Événements avec des abonnés asynchrones

Vous avez un modèle final à découvrir : comment écrire correctement des abonnés à l'événement qui appellent du code asynchrone. La difficulté est décrite dans l'article consacré à async et await. Les méthodes asynchrones peuvent avoir un type de retour void, mais ceci est fortement déconseillé. Quand votre code pour l'abonné à l'événement appelle une méthode asynchrone, vous n'avez d'autre choix que de créer une méthode `async void`. La signature de gestionnaire d'événements en a besoin.

Vous devez rapprocher ces deux nécessités opposées. D'une façon ou d'une autre, vous devez créer une méthode `async void` sécurisée. Les principes de base du modèle que vous devez implémenter sont les suivants :

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

Notez d'abord que le gestionnaire est marqué en tant que gestionnaire asynchrone. Comme il est affecté à un type délégué de gestionnaire d'événements, il a un type de retour void. Cela signifie que vous devez suivre le modèle indiqué dans le gestionnaire et interdire la levée d'exceptions en dehors du contexte du gestionnaire asynchrone. Comme il ne retourne pas une tâche, aucune tâche ne peut signaler l'erreur en passant à l'état d'erreur. Comme la méthode est asynchrone, elle ne peut tout simplement pas lever l'exception. (La méthode d'appel a `async` continué l'exécution parce qu'elle est .) Le comportement réel en temps d'exécution sera défini différemment pour différents environnements. Il peut mettre fin au thread ou au processus qui possède le thread, ou laisser le processus dans un état indéterminé. Tous ces résultats potentiels sont très indésirables.

C'est pourquoi vous devez encapsuler l'instruction await pour la tâche asynchrone dans votre propre bloc try. Si elle entraîne une erreur dans une tâche, vous pouvez consigner cette erreur. Si c'est une erreur dont votre application ne peut pas récupérer, vous pouvez quitter le programme rapidement et normalement.

Ce sont les principales mises à jour du modèle d'événement .NET. Vous pouvez voir de nombreux exemples des versions précédentes dans les bibliothèques avec lesquelles vous travaillez. Il est cependant important de bien comprendre ce que sont les derniers modèles.

Le prochain article de cette série vous permet de faire la distinction entre l'utilisation de `delegates` et de `events` dans vos conceptions. Ces sont des concepts similaires : cet article vous aide à prendre la bonne décision pour vos programmes.

Suivant

Différenciation des délégués et des événements

02/11/2020 • 7 minutes to read • Edit Online

Précédent

Les développeurs qui découvrent la plateforme .NET Core éprouvent souvent des difficultés quand il faut choisir entre une conception basée sur les `delegates` et une conception basée sur les `events`. Le choix des délégués ou des événements est souvent difficile, car les deux fonctionnalités de langage sont similaires. Les événements sont même générés à l'aide de la prise en charge linguistique des délégués.

Elles offrent tous deux un scénario de liaison tardive : elles autorisent les scénarios où un composant communique en appelant une méthode qui est connue uniquement au moment de l'exécution. Toutes deux prennent en charge les méthodes à abonnés uniques et multiples. On utilise parfois le terme de prise en charge singlecast et multicast. Toutes deux prennent en charge une syntaxe similaire pour l'ajout et la suppression de gestionnaires. Pour finir, le déclenchement d'un événement et l'appel d'un délégué utilisent exactement la même syntaxe d'appel de méthode. Qui plus est, elles prennent toutes deux en charge la même syntaxe de méthode `Invoke()` pour une utilisation avec l'opérateur `?.`.

Avec tous ces similitudes, il est facile d'avoir des difficultés à déterminer quand utiliser l'une ou l'autre.

La détection d'événements est facultative

Pour déterminer laquelle de ces deux fonctionnalités utiliser, il convient avant tout de savoir s'il doit y avoir un abonné attaché ou non. Si votre code doit appeler le code fourni par l'abonné, vous devez utiliser une conception basée sur des délégués lorsque vous devez implémenter le rappel. Si votre code peut effectuer tout son travail sans appeler les abonnés, vous devez utiliser un modèle basé sur les événements.

Examinez les exemples fournis dans cette section. Le code que vous avez créé à l'aide de `List.Sort()` doit disposer d'une fonction de comparaison pour trier correctement les éléments. Des requêtes LINQ doivent être fournies avec les délégués pour identifier les éléments à retourner. Tous deux ont une conception basée sur des délégués.

Prenez l'événement `Progress`. Il signale la progression d'une tâche. La tâche se poursuit même s'il n'y a aucun détecteur. `FileSearcher` constitue un autre exemple. Il trouvera tous les fichiers recherchés même si aucun abonné aux événements n'est attaché. Les contrôles de l'expérience utilisateur fonctionneront toujours correctement, même si aucun abonné n'est à l'écoute des événements. Ils utilisent tous deux des conceptions basées sur les événements.

Les valeurs de retour nécessitent des délégués

Une autre considération à prendre en compte est le prototype de méthode souhaité pour votre méthode déléguée. Comme nous l'avons vu, les délégués utilisés pour les événements ont tous un type de retour void. Nous avons également vu qu'il existait des idiomes pour créer des gestionnaires d'événements qui repassent des informations aux sources d'événements en modifiant des propriétés de l'objet d'argument d'événement. Bien que ces idiomes fonctionnent, ils ne sont pas aussi naturels que le retour d'une valeur à partir d'une méthode.

Notez que ces heuristiques peuvent souvent être toutes deux présentes : si votre méthode déléguée retourne une valeur, elle aura probablement un impact sur l'algorithme.

Les événements ont un appel privé

Les classes autres que celle dans laquelle un événement est contenu peuvent uniquement ajouter et supprimer des écouteurs d'événements. seule la classe contenant l'événement peut appeler l'événement. Les événements sont

généralement des membres de classe publics. Par comparaison, les délégués sont souvent passés comme paramètres et stockés comme membres de classe privée, s'ils sont stockés du tout.

Les détecteurs d'événements ont souvent des durées de vie plus longues

Ces écouteurs d'événements ont une durée de vie légèrement plus faible. Toutefois, vous constaterez peut-être que les conceptions basées sur les événements sont plus naturelles quand la source de l'événement déclenche des événements durant une période prolongée. Vous pouvez voir des exemples de conception basée sur les événements pour les contrôles d'expérience utilisateur sur de nombreux systèmes. Une fois que vous vous êtes abonné à un événement, la source de l'événement peut déclencher des événements pendant toute la durée de vie du programme. (Vous pouvez vous désabonner à des événements quand vous n'en avez plus besoin.)

Comparez cela à de nombreuses conceptions basées sur les délégués, où un délégué est utilisé comme argument d'une méthode et le délégué n'est pas utilisé après le retour de cette méthode.

Évaluez avec soin

Les considérations ci-dessus ne sont pas des règles absolues. Il s'agit plutôt de conseils qui vous aideront à décider de ce qui convient le mieux à votre utilisation particulière. Les deux fonctionnalités étant similaires, vous pouvez même les prototyper et évaluer celle dont l'utilisation serait plus naturelle. Elles gèrent également toutes deux les scénarios de liaison tardive. Utilisez celle qui correspond le mieux à votre conception.

LINQ (Language-Integrated Query)

02/11/2020 • 6 minutes to read • Edit Online

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.
- Les expressions de requête sont faciles à maîtriser, car elles utilisent de nombreuses constructions familières du langage C#.
- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de

nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.
- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.
- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.
- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez syntaxe de requête et syntaxe de méthode dans LINQ.
- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Concepts de base des expressions de requête

18/03/2020 • 22 minutes to read • Edit Online

Cet article présente les concepts fondamentaux liés aux expressions de requête en C#.

Qu'est-ce qu'une requête et quel est son rôle ?

Une *requête* est un jeu d'instructions qui décrit quelles données doivent être récupérées à partir d'une source (ou de sources) de données fournie, et quelles forme et organisation les données retournées doivent avoir. Une requête est distincte des résultats qu'elle génère.

De manière générale, les données sources sont organisées logiquement comme une séquence d'éléments du même type. Par exemple, une table de base de données SQL contient une séquence de lignes. Un fichier XML contient une « séquence » d'éléments XML (bien que ceux-ci soient organisés hiérarchiquement dans une arborescence). Une collection en mémoire contient une séquence d'objets.

Du point de vue d'une application, le type et la structure spécifiques des données sources d'origine ne sont pas importants. L'application voit toujours les données sources comme une collection IEnumerable<T> ou IQueryable<T>. Par exemple, dans LINQ to XML, les données sources sont rendues visibles sous la forme d'un `IEnumerable` <XElement.

Compte tenu de cette séquence source, une requête peut effectuer l'une des trois actions suivantes :

- Récupérer un sous-ensemble des éléments pour générer une nouvelle séquence sans modifier les éléments individuels. La requête peut alors trier ou regrouper la séquence retournée de plusieurs façons, comme indiqué dans l'exemple suivant (supposons que `scores` est un `int[]`) :

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Récupérer une séquence d'éléments comme dans l'exemple précédent, mais les transformer en un nouveau type d'objet. Par exemple, une requête peut récupérer uniquement les noms de certains enregistrements de clients dans une source de données. Elle peut aussi récupérer l'enregistrement complet, puis l'utiliser pour construire un autre type d'objet en mémoire, voire des données XML, avant de générer la séquence de résultat final. L'exemple suivant affiche une projection d'un `int` en `string`. Notez le nouveau type de `highScoresQuery`.

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Récupérer une valeur singleton sur les données sources, par exemple :
  - Nombre d'éléments qui correspondent à une certaine condition.
  - Élément qui a la valeur la plus grande ou la plus petite.
  - Premier élément qui correspond à une condition, ou somme de valeurs particulières dans un jeu

d'éléments spécifié. Par exemple, la requête suivante retourne le nombre de notes supérieures à 80 à partir du tableau d'entiers `scores` :

```
int highScoreCount =
    (from score in scores
     where score > 80
     select score)
     .Count();
```

Dans l'exemple précédent, notez l'utilisation de parenthèses autour de l'expression de requête avant l'appel à la méthode `Count`. Vous pouvez également exprimer ceci en utilisant une nouvelle variable pour stocker le résultat concret. Cette technique est plus explicite, car elle conserve la variable qui stocke la requête à un emplacement distinct de la requête qui stocke un résultat.

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

Dans l'exemple précédent, la requête est exécutée dans l'appel à `Count`, car `Count` doit effectuer une itération sur les résultats afin de déterminer le nombre d'éléments retournés par `highScoresQuery`.

Qu'est-ce qu'une expression de requête ?

Une *expression de requête* est une requête exprimée dans la syntaxe de la requête. Une expression de requête est une construction de langage de premier ordre. Elle est similaire à toute autre expression et peut être utilisée dans tous les contextes dans lesquels une expression C# est valide. Une expression de requête se compose d'un jeu de clauses écrit dans une syntaxe déclarative semblable à du SQL ou du XQuery. Chaque clause contient à son tour une ou plusieurs expressions C#, et ces expressions peuvent elles-mêmes être une expression de requête ou contenir une expression de requête.

Une expression de requête doit commencer par une clause from et doit se terminer par une clause select ou group. Entre la première clause `from` et la dernière clause `select` ou `group`, elle peut contenir une ou plusieurs clauses facultatives : where, orderby, join, let et même d'autres clauses from. Vous pouvez également utiliser le mot clé into pour que le résultat d'une clause `join` ou `group` puisse servir de source pour des clauses de requête supplémentaires dans la même expression de requête.

Variable de requête

Dans LINQ, une variable de requête correspond à n'importe quelle variable qui stocke une *requête* au lieu des *résultats* d'une requête. Plus spécifiquement, une variable de requête est toujours un type énumérable qui produit une séquence d'éléments quand elle est itérée dans une instruction `foreach` ou un appel direct à sa méthode `IEnumerator.MoveNext`.

L'exemple de code suivant montre une expression de requête simple avec une source de données, une clause de filtrage, une clause de classement et aucune transformation des éléments sources. La clause `select` termine la requête.

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

Dans l'exemple précédent, `scoreQuery` est une *variable de requête* qui est parfois désignée simplement sous le nom de *requête*. La variable de requête ne stocke aucune donnée de résultat réelle, générée dans la boucle `foreach`. De plus, quand l'instruction `foreach` s'exécute, les résultats de la requête ne sont pas retournés via la variable de requête `scoreQuery`. Au lieu de cela, ils sont retournés via la variable d'itération `testScore`. La variable `scoreQuery` peut être itérée dans une deuxième boucle `foreach`. Elle générera les mêmes résultats tant que ni elle ni la source de données ne sont modifiées.

Une variable de requête peut stocker une requête exprimée dans la syntaxe de requête ou dans la syntaxe de méthode, ou dans une combinaison des deux. Dans les exemples suivants, `queryMajorCities` et `queryMajorCities2` sont des variables de requête :

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);
```

En revanche, les deux exemples suivants affichent des variables qui ne sont pas des variables de requête, même si chacune d'entre elles est initialisée avec une requête. Il ne s'agit pas de variables de requête, car elles stockent des résultats :

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

Pour plus d'informations sur les différents modes d'expression de requêtes, consultez Syntaxe de requête et syntaxe de méthode dans LINQ.

Types explicites et implicites de variables de requête

Cette documentation fournit généralement le type explicite de la variable de requête pour afficher la relation de type entre la variable de requête et la clause select. Toutefois, vous pouvez également utiliser le mot clé var pour indiquer au compilateur de déduire le type d'une variable de requête (ou toute autre variable locale) au moment de la compilation. Par exemple, l'exemple de requête expliqué précédemment dans cette rubrique peut également être exprimé à l'aide de types implicites :

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

Pour plus d'informations, consultez Variables locales implicitement typées et Relations des types dans des opérations de requête LINQ.

Démarrage d'une expression de requête

Une expression de requête doit commencer par une clause `from`. Elle spécifie une source de données avec une variable de portée. La variable de portée représente chaque élément consécutif dans la séquence source comme la séquence source parcourue. La variable de portée est fortement typée selon le type d'éléments dans la source de données. Dans l'exemple suivant, comme `countries` est un tableau d'objets `Country`, la variable de portée est également de type `Country`. Étant donné que la variable de portée est fortement typée, vous pouvez utiliser l'opérateur point pour accéder à tous les membres disponibles du type.

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

La variable de portée est dans la portée tant que la requête n'est pas quittée avec un point-virgule ou une clause *continuation*.

Une expression de requête peut contenir plusieurs clauses `from`. Utilisez des clauses `from` supplémentaires quand chaque élément de la séquence source est lui-même une collection ou contient une collection. Par exemple, supposons que vous avez une collection d'objets `Country` et que chacun d'entre eux contient une collection d'objets `City` nommés `Cities`. Pour interroger les objets `City` dans chaque `Country`, utilisez deux clauses `from` comme indiqué ici :

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Pour plus d'informations, consultez from, clause.

Fin d'une expression de requête

Une expression de requête doit se terminer par une clause `group` ou une clause `select`.

group, clause

Utilisez la clause `group` pour générer une séquence de groupes organisée par une clé que vous spécifiez. La clé peut correspondre à tout type de données. Par exemple, la requête suivante crée une séquence de groupes qui contient un ou plusieurs objets `Country` dont la clé est une valeur `char`.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Pour plus d'informations sur le regroupement, consultez group, clause.

select, clause

Utilisez la clause `select` pour générer tous les autres types de séquences. Une clause `select` simple génère simplement une séquence du même type d'objets que les objets contenus dans la source de données. Dans cet exemple, la source de données contient des objets `Country`. La clause `orderby` trie simplement les éléments dans un nouvel ordre et la clause `select` génère une séquence des objets `Country` réorganisés.

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

La clause `select` peut être utilisée pour transformer des données sources en séquences de nouveaux types. Cette transformation est également appelée *projection*. Dans l'exemple suivant, la clause `select` *projette* une séquence de types anonymes qui contient uniquement un sous-ensemble des champs dans l'élément d'origine. Notez que les nouveaux objets sont initialisés à l'aide d'un initialiseur d'objet.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

Pour plus d'informations sur toutes les façons d'utiliser une clause `select` pour transformer des données sources, consultez select, clause.

Continuations avec « into »

Vous pouvez utiliser le mot clé `into` dans une clause `select` ou `group` pour créer un identificateur temporaire qui stocke une requête. Procédez ainsi quand vous devez effectuer des opérations de requête supplémentaires sur une requête après une opération de regroupement ou de sélection. Dans l'exemple suivant, les `countries` sont regroupés en fonction du remplissage, par plages de 10 millions. Une fois ces groupes créés, des clauses supplémentaires éliminent des groupes par filtrage, puis trient les groupes par ordre croissant. Pour effectuer ces opérations supplémentaires, la continuation représentée par `countryGroup` est requise.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

Pour plus d'informations, consultez into.

Filtrage, classement et jointure

Entre la clause `from` initiale et la clause `select` ou `group` finale, toutes les autres clauses (`where`, `join`, `orderby`, `from`, `let`) sont facultatives. Chacune des clauses facultatives peut être utilisée zéro fois ou plusieurs fois dans un corps de requête.

where, clause

Utilisez la clause `where` pour éliminer par filtrage des éléments des données sources selon une ou plusieurs expressions de prédicat. La clause `where` de l'exemple suivant a un prédicat avec deux conditions.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Pour plus d'informations, consultez where, clause.

orderby, clause

Utilisez la clause `orderby` pour trier les résultats par ordre croissant ou décroissant. Vous pouvez également spécifier des ordres de tri secondaires. L'exemple suivant effectue un tri principal sur les objets `country` en utilisant la propriété `Area`. Il effectue ensuite un tri secondaire en utilisant la propriété `Population`.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

Le mot clé `ascending` est facultatif ; il s'agit de l'ordre de tri par défaut si aucun ordre n'est spécifié. Pour plus d'informations, voir clause d'ordre.

join, clause

Utilisez la clause `join` pour associer et/ou combiner des éléments d'une source de données avec des éléments d'une autre source de données en fonction d'une comparaison d'égalité entre des clés spécifiées dans chaque élément. Dans LINQ, les opérations de jointure sont effectuées sur les séquences des objets dont les éléments sont des types différents. Après avoir joint deux séquences, vous devez utiliser une instruction `select` ou `group` pour spécifier quel élément stocker dans la séquence de sortie. Vous pouvez également utiliser un type anonyme pour combiner des propriétés de chaque jeu d'éléments associés dans un nouveau type pour la séquence de sortie. L'exemple suivant associe des objets `prod` dont la propriété `Category` correspond à l'une des catégories dans le tableau de chaînes `categories`. Les `Category` produits qui ne `categories` correspondent à aucune chaîne sont filtrés. L'énoncé `select` projette un nouveau type `cat` `prod` dont les propriétés sont prises à la fois et .

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

Vous pouvez également effectuer une jointure groupée en stockant les résultats de l'opération `join` dans une variable temporaire à l'aide du mot clé into. Pour plus d'informations, voir clause d'adhésion.

clause let

Utilisez la clause `let` pour stocker le résultat d'une expression telle qu'un appel de méthode dans une nouvelle variable de portée. Dans l'exemple suivant, la variable de portée `firstName` stocke le premier élément du tableau de chaînes retourné par `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

Pour plus d'informations, consultez let, clause.

Sous-requêtes dans une expression de requête

Une clause de requête peut elle-même contenir une expression de requête, qui est parfois connue sous le nom de *sous-requête*. Chaque sous-requête démarre avec sa propre clause `from` qui ne pointe pas nécessairement vers la même source de données dans la première clause `from`. Par exemple, la requête suivante montre une expression de requête utilisée dans l'instruction select pour récupérer les résultats d'une opération de regroupement.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

Pour plus d'informations, voir Effectuer une sous-querie sur une opération de regroupement.

Voir aussi

- Guide de programmation CMD
- Requête intégrée linguistique (LINQ)
- Mots clés de requête (LINQ)
- Aperçu standard des opérateurs de requêtes

LINQ en C#

18/03/2020 • 2 minutes to read • Edit Online

Cette section contient des liens vers des rubriques qui fournissent des informations plus détaillées à propos de LINQ.

Contenu de cette section

Introduction aux requêtes LINQ
Décrit les trois parties de l'opération de requête LINQ de base qui sont communes à tous les langages et à toutes les sources de données.

LINQ et types génériques
Fournit une brève introduction aux types génériques utilisés dans LINQ.

Transformations de données avec LINQ
Décrit les différentes façons dont vous pouvez transformer les données récupérées dans des requêtes.

Relations des types dans des opérations de requête LINQ
Décrit comment les types sont préservés et/ou transformés dans les trois parties d'une opération de requête LINQ

Syntaxe de requête et syntaxe de méthode dans LINQ
Compare la syntaxe des méthodes et la syntaxe des requêtes considérées comme deux moyens d'exprimer une requête LINQ.

Fonctionnalités C# qui prennent en charge LINQ
Décrit les constructions de langage dans C# qui prennent en charge LINQ.

Sections connexes

Expressions de requête LINQ
Inclut une vue d'ensemble des requêtes dans LINQ et fournit des liens vers des ressources supplémentaires.

Aperçu standard des opérateurs de requêtes
Présente les méthodes standard utilisées dans LINQ.

Écrire des requêtes LINQ en C#

02/11/2020 • 7 minutes to read • Edit Online

Cet article présente les trois façons d'écrire une requête LINQ en C# :

1. Utilisez la syntaxe de requête.
2. Utilisez la syntaxe de méthode.
3. Utilisez une combinaison de syntaxe de requête et de syntaxe de méthode.

Les exemples suivants montrent des requêtes LINQ simples en utilisant chaque approche répertoriée précédemment. En général, la règle consiste à utiliser (1) dès que possible et à utiliser (2) et (3) dès que cela est nécessaire.

> **NOTE**
> Ces requêtes fonctionnent sur les collections simples en mémoire ; toutefois, la syntaxe de base est identique à celle utilisée dans LINQ to Entities et LINQ to XML.

Exemple – Syntaxe de requête

La méthode recommandée pour écrire la plupart des requêtes consiste à utiliser la *syntaxe de requête* pour créer des *expressions de requête*. L'exemple suivant présente trois expressions de requête. La première expression de requête montre comment filtrer ou restreindre des résultats en appliquant des conditions avec une clause `where`. Tous les éléments de la séquence source dont la valeur est supérieure à 7 ou inférieure à 3 sont retournés. La deuxième expression montre comment classer les résultats retournés. La troisième expression montre comment regrouper des résultats en fonction d'une clé. Cette requête retourne deux groupes en fonction de la première lettre du mot.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Notez que le type des requêtes est IEnumerable<T>. Toutes ces requêtes pourraient être écrites à l'aide de `var`, comme illustré dans l'exemple suivant :

```
var query = from num in numbers...
```

Dans chacun des exemples précédents, les requêtes ne s'exécutent pas réellement tant vous n'avez pas itéré la variable de requête dans une instruction `foreach` ou une autre instruction. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

Exemple – Syntaxe de méthode

Certaines opérations de requête doivent être exprimées comme un appel de méthode. Les plus répandues de ces méthodes retournent des valeurs numériques uniques, telles que Sum, Max, Min, Average, et ainsi de suite. Ces méthodes doivent toujours être appelées en dernier dans toutes les requêtes, car elles ne représentent qu'une valeur unique et ne peuvent pas servir de source pour une opération de requête supplémentaire. L'exemple suivant présente un appel de méthode dans une expression de requête :

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Si la méthode a des paramètres Action ou Func, ceux-ci sont fournis sous la forme d'une expression lambda, comme dans l'exemple suivant :

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Dans les requêtes précédentes, seule la requête 4 s'exécute immédiatement. Cela s'explique par le fait qu'elle retourne une valeur unique et non une collection IEnumerable<T> générique. La méthode elle-même doit utiliser `foreach` pour calculer sa valeur.

Chacune des requêtes précédentes peut être écrite en utilisant des types implicites avec var, comme dans l'exemple suivant :

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Exemple – Syntaxe de méthode et de requête mixte

Cet exemple montre comment utiliser la syntaxe de méthode sur les résultats d'une clause de requête. Encadrez simplement l'expression de requête entre parenthèses, puis appliquez l'opérateur point et appelez la méthode. Dans l'exemple suivant, la requête 7 retourne les nombres dont la valeur est comprise entre 3 et 7. Toutefois, il est généralement préférable d'utiliser une deuxième variable pour stocker le résultat de l'appel de méthode. De cette manière, il est moins probable que la requête ne soit confondue avec les résultats de la requête.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Comme la requête 7 retourne une valeur unique et non une collection, la requête s'exécute immédiatement.

La requête précédente peut être écrite en utilisant des types implicites avec `var`, comme suit :

```
var numCount = (from num in numbers...
```

Elle peut être écrite dans la syntaxe de méthode comme suit :

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Elle peut être écrite en utilisant des types explicites, comme suit :

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Voir aussi

- Procédure pas à pas : écriture de requêtes en C#
- LINQ (Language-Integrated Query)
- clause WHERE

Interroger une collection d'objets

18/03/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment effectuer une requête simple sur une liste d'objets `Student`. Chaque objet `Student` contient des informations de base sur l'étudiant et une liste qui représente les notes de l'étudiant lors de quatre examens.

Cette application sert de cadre pour de nombreux autres exemples dans cette section qui utilisent la même source de données `students`.

Exemple

La requête suivante retourne les étudiants qui ont reçu une note supérieure ou égale à 90 lors de leur premier examen.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
```

```
            ExamScores = new List<int> { 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 96, 85, 91, 60}}
    };
    #endregion

    // Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public static void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Cette requête est volontairement simple pour vous permettre de faire des essais. Par exemple, vous pouvez essayer plus de conditions dans la clause `where` ou utiliser une clause `orderby` pour trier les résultats.

Voir aussi

- Requête intégrée linguistique (LINQ)
- Interpolation de chaîne

Comment retourner une requête à partir d'une méthode (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment retourner une requête à partir d'une méthode en tant que valeur de retour et en tant que paramètre `out`.

Les objets de requête sont composables, ce qui signifie que vous pouvez retourner une requête à partir d'une méthode. Les objets qui représentent des requêtes ne stockent pas la collection résultante, mais plutôt les étapes pour générer les résultats lorsque cela est nécessaire. L'avantage de retourner des objets de requête à partir de méthodes est qu'il est possible de les composer ou modifier encore. Par conséquent, toute valeur de retour ou paramètre `out` d'une méthode qui retourne une requête doit également avoir ce type. Si une méthode matérialise une requête en un type List<T> ou Array concret, elle est considérée comme retournant les résultats de la requête plutôt que la requête elle-même. Une variable de requête retournée à partir d'une méthode peut encore être composée ou modifiée.

Exemple

Dans l'exemple suivant, la première méthode retourne une requête comme valeur de retour et la seconde méthode retourne une requête comme paramètre `out`. Notez que, dans les deux cas, une requête est retournée et non pas les résultats d'une requête.

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
```

```
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }

        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out parameter.
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }

        // You can modify a query by using query composition. A saved query
        // is nested inside a new query definition that revises the results
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1:");
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Voir aussi

- LINQ (Language-Integrated Query)

Stocker les résultats d'une requête dans la mémoire

18/03/2020 • 2 minutes to read • Edit Online

Une requête est en fait un ensemble d'instructions permettant de récupérer et d'organiser des données. Les requêtes sont exécutées de manière différée, à mesure que chaque élément dans le résultat est demandé. Lorsque vous utilisez `foreach` pour effectuer une itération sur les résultats, les éléments sont retournés lorsque vous y accédez. Pour évaluer une requête et stocker ses résultats sans exécuter une boucle `foreach`, appelez simplement l'une des méthodes suivantes sur la variable de requête :

- ToList
- ToArray
- ToDictionary
- ToLookup

Lorsque vous stockez les résultats de requête, nous vous recommandons d'assigner l'objet de collection retourné à une nouvelle variable, comme indiqué dans l'exemple suivant :

Exemple

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

Voir aussi

- Requête intégrée linguistique (LINQ)

Regrouper les résultats d'une requête

18/03/2020 • 10 minutes to read • Edit Online

Le regroupement est l'une des fonctionnalités les plus puissantes de LINQ. Les exemples suivants montrent comment regrouper des données de différentes manières :

- Selon une propriété
- Selon la première lettre d'une propriété de chaîne
- Selon une plage numérique calculée
- Selon un prédicat booléen ou une autre expression
- Selon une clé composée

En outre, les deux dernières requêtes projettent leurs résultats dans un nouveau type anonyme qui contient uniquement le prénom et le nom de l'étudiant. Pour plus d'informations, consultez group, clause.

Exemple

Tous les exemples de cette rubrique utilisent les classes d'assistance et les sources de données suivantes.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
```

```
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Exemple

L'exemple suivant montre comment regrouper des éléments source en utilisant une propriété de l'élément comme clé de groupe. Dans ce cas, la clé est un `string`, qui correspond au nom de l'étudiant. Il est également possible d'utiliser une sous-chaîne comme clé. L'opération de regroupement utilise le comparateur d'égalité par défaut pour le type.

Copiez-collez la méthode suivante dans la classe `StudentClass`. Remplacez l'instruction d'appel de la méthode `Main` par `sc.GroupBySingleProperty()`.

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Exemple

L'exemple suivant montre comment regrouper des éléments source en utilisant autre chose qu'une propriété de l'objet comme clé de groupe. Dans cet exemple, la clé est la première lettre du nom de l'étudiant.

Copiez-collez la méthode suivante dans la classe `StudentClass`. Remplacez l'instruction d'appel de la méthode `Main` par `sc.GroupBySubstring()`.

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Exemple

L'exemple suivant montre comment regrouper des éléments source en utilisant une plage numérique comme clé de groupe. La requête projette ensuite les résultats dans un type anonyme qui contient uniquement le prénom et le nom de l'étudiant, ainsi que la plage de centiles à laquelle appartient l'étudiant. Un type anonyme est utilisé, car il n'est pas nécessaire d'utiliser l'intégralité de l'objet `Student` pour afficher les résultats. `GetPercentile` est une fonction d'assistance qui calcule un centile à partir de la note moyenne obtenue par l'étudiant. La méthode retourne un entier compris entre 0 et 10.

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

Copiez-collez la méthode suivante dans la classe `StudentClass`. Remplacez l'instruction d'appel de la méthode `Main` par `sc.GroupByRange()`.

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Exemple

L'exemple suivant montre comment regrouper des éléments source en utilisant une expression de comparaison booléenne. Dans cet exemple, l'expression booléenne teste si la note moyenne d'un étudiant est supérieure à 75. Comme dans les exemples précédents, les résultats sont projetés dans un type anonyme, car il n'est pas nécessaire d'avoir l'élément source complet. Notez que les propriétés du type anonyme deviennent des propriétés du membre `Key` et sont accessibles par nom lorsque la requête est exécutée.

Copiez-collez la méthode suivante dans la classe `StudentClass`. Remplacez l'instruction d'appel de la méthode `Main` par `sc.GroupByBoolean()`.

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Exemple

L'exemple suivant montre comment utiliser un type anonyme pour encapsuler une clé qui contient plusieurs valeurs. Dans cet exemple, la première valeur de clé est la première lettre du nom de l'étudiant. La deuxième valeur de clé est une valeur booléenne qui spécifie si l'étudiant a obtenu une note supérieure à 85 au premier examen. Vous pouvez organiser les groupes selon n'importe quelle propriété de la clé.

Copiez-collez la méthode suivante dans la classe `StudentClass`. Remplacez l'instruction d'appel de la méthode `Main` par `sc.GroupByCompositeKey()`.

```
public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"\t{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

Voir aussi

- GroupBy
- IGrouping<TKey,TElement>
- Requête intégrée linguistique (LINQ)
- group, clause
- Types anonymes
- Effectuer une sous-querie sur une opération de regroupement
- Créer un groupe imbriqué
- Regrouper les données

Créer un groupe imbriqué

18/03/2020 • 2 minutes to read • Edit Online

L'exemple suivant montre comment créer des groupes imbriqués dans une expression de requête LINQ. Chaque groupe créé en fonction de l'année/du niveau d'étude est ensuite encore subdivisé en groupes en fonction des noms des individus.

Exemple

> **NOTE**
>
> Cet exemple contient des références aux objets définis dans l'exemple de code qui est présenté dans Interroger une collection d'objets.

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"\t\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

Notez que trois boucles `foreach` imbriquées sont nécessaires pour effectuer une itération sur les éléments internes d'un groupe imbriqué.

Voir aussi

- Requête intégrée linguistique (LINQ)

Effectuer une sous-requête sur une opération de regroupement

18/03/2020 • 2 minutes to read • Edit Online

Cet article présente deux façons de créer une requête qui organise les données sources en groupes et effectue ensuite une sous-requête sur chacun de ces groupes. Dans chaque exemple, la technique de base consiste à regrouper les éléments sources en utilisant une *continuation* nommée `newGroup`, puis en créant une sous-requête sur `newGroup`. Cette sous-requête est exécutée sur chaque groupe créé par la requête externe. Notez que, dans cet exemple particulier, le résultat final n'est pas un groupe, mais une séquence plate de types anonymes.

Pour plus d'informations sur les regroupements, consultez group, clause.

Pour plus d'informations sur les continuations, consultez into. L'exemple suivant utilise une structure de données en mémoire comme source de données, mais les mêmes principes s'appliquent à tous les types de sources de données LINQ.

Exemple

NOTE

Cet exemple contient des références aux objets définis dans l'exemple de code qui est présenté dans Interroger une collection d'objets.

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

La requête de l'extrait de code ci-dessus peut également s'écrire avec la syntaxe de méthode. L'extrait de code suivant comporte une requête sémantiquement équivalente, écrite avec la syntaxe de méthode.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

Voir aussi

- Requête intégrée linguistique (LINQ)

Regrouper des résultats par clés contiguës

18/03/2020 • 8 minutes to read • Edit Online

L'exemple suivant montre comment regrouper des éléments dans des blocs représentant des sous-séquences de clés contiguës. Par exemple, supposons que vous disposiez de la séquence de paires clé-valeur suivante :

CLÉ	VALEUR
Un	Nous
Un	pensons
Un	que
B	Linq
C	is
Un	vraiment
B	chouette
B	!

Les groupes suivants seront créés dans cet ordre :

1. Nous, pensons, que
2. Linq
3. is
4. vraiment
5. chouette, !

La solution est implémentée comme une méthode d'extension thread-safe qui retourne ses résultats en continu. En d'autres termes, elle produit ses groupes à mesure qu'elle se déplace dans la séquence source. Contrairement aux opérateurs `group` ou `orderby`, elle peut commencer à retourner des groupes à l'appelant avant que l'intégralité de la séquence ait été lue.

La cohérence de thread est obtenue en effectuant une copie de chaque groupe ou bloc pendant l'itération de la séquence source, comme expliqué dans les commentaires du code source. Si la séquence source comporte une longue séquence d'éléments contigus, le common language runtime peut lever une OutOfMemoryException.

Exemple

L'exemple suivant montre la méthode d'extension et le code client qui l'utilise :

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

            // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk. Each
```

```
ChunkItem
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }

            // The value that is used to determine matching elements
            private readonly TKey key;

            // Stores a reference to the enumerator for the source sequence
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare keys.
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the end of the source sequence.
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the list contains > 1 elements.
                tail = head;

                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied to the list of ChunkItems,
            // and the source enumerator is either at the end, or else on an element with a new key.
            // the tail of the linked list is set to null in the CopyNextChunkElement method if the
            // key of the next element does not match the current chunk's key, or there are no more elements in
the source.
            private bool DoneCopyingChunk => tail == null;

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequence.
                // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) at the end of the current chunk
                // then null out the enumerator and predicate for reuse with the next chunk.
                if (isLastSourceElement || !predicate(enumerator.Current))
```

```
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next;
    }

    // Called after the end of the last chunk was reached. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumerator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.
        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }
```

```
            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
        }
    }

    // A simple named type is used for easier viewing in the debugger. Anonymous types
    // work just as well with the ChunkBy operator.
    public class KeyValPair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    class Program
    {
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;

        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initializer.
            list = new[]
            {
                new KeyValPair{ Key = "A", Value = "We" },
                new KeyValPair{ Key = "A", Value = "think" },
                new KeyValPair{ Key = "A", Value = "that" },
                new KeyValPair{ Key = "B", Value = "Linq" },
                new KeyValPair{ Key = "C", Value = "is" },
                new KeyValPair{ Key = "A", Value = "really" },
                new KeyValPair{ Key = "B", Value = "cool" },
                new KeyValPair{ Key = "B", Value = "!" }
            };

            // Create the query by using our user-defined query operator.
            query = list.ChunkBy(p => p.Key);

            // ChunkBy returns IGrouping objects, therefore a nested
            // foreach loop is required to access the elements in each "chunk".
            foreach (var item in query)
            {
                Console.WriteLine($"Group key = {item.Key}");
                foreach (var inner in item)
                {
                    Console.WriteLine($"\t{inner.Value}");
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

Pour utiliser la méthode d'extension dans votre projet, copiez la classe statique `MyExtensions` dans un fichier de code source nouveau ou existant et, si nécessaire, ajoutez une directive `using` pour l'espace de noms dans lequel elle se trouve.

Voir aussi

- Requête intégrée linguistique (LINQ)

Spécifier dynamiquement des filtres de prédicat au moment de l'exécution

18/03/2020 • 4 minutes to read • Edit Online

Dans certains cas, ce n'est qu'au moment de l'exécution que vous savez combien de prédicats vous devez appliquer aux éléments sources dans la clause `where`. L'une des manières de spécifier plusieurs filtres de prédicat de manière dynamique consiste à utiliser la méthode Contains, comme indiqué dans l'exemple suivant. L'exemple est construit de deux façons. Tout d'abord, le projet est exécuté en filtrant les valeurs fournies dans le programme. Ensuite, le projet est réexécuté à l'aide de l'entrée fournie au moment de l'exécution.

Pour filtrer à l'aide de la méthode Contains

1. Ouvrez une nouvelle application console et nommez-la `PredicateFilters`.
2. Copiez la classe `StudentClass` à partir de Interroger une collection d'objets et collez-la dans l'espace de noms `PredicateFilters` sous la classe `Program`. `StudentClass` fournit une liste d'objets `Student`.
3. Commentez la méthode `Main` dans `StudentClass`.
4. Remplacez la classe `Program` par le code suivant :

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Ajoutez la ligne suivante à la méthode `Main` dans la classe `DynamicPredicates`, sous la déclaration de `ids`.

```
QueryById(ids);
```

6. Exécutez le projet.
7. La sortie suivante s'affiche dans une fenêtre de console :

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. L'étape suivante consiste à réexécuter le projet, cette fois en utilisant l'entrée saisie au moment de l'exécution plutôt que le tableau `ids`. Remplacez `QueryByID(ids)` par `QueryByID(args)` dans la méthode `Main`.

9. Exécutez le projet avec les arguments de ligne de commande `122 117 120 115`. Quand le projet s'exécute, ces valeurs deviennent des éléments de `args`, le paramètre de la méthode `Main`.

10. La sortie suivante s'affiche dans une fenêtre de console :

    Adams: 120

    Feng: 117

    Garcia: 115

    Tucker: 122

Pour filtrer à l'aide d'une instruction switch

1. Vous pouvez utiliser une instruction `switch` pour sélectionner parmi d'autres requêtes prédéterminées. Dans l'exemple suivant, `studentQuery` utilise une autre clause `where` en fonction du niveau, ou de l'année, spécifié(e) au moment de l'exécution.

2. Copiez la méthode suivante et collez-la dans la classe `DynamicPredicates`.

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. Dans la méthode `Main`, remplacez l'appel à `QueryByID` par l'appel suivant, qui envoie le premier élément du tableau `args` comme argument : `QueryByYear(args[0])`.

4. Exécutez le projet avec un argument de ligne de commande d'une valeur entière comprise entre 1 et 4.

Voir aussi

- Requête intégrée linguistique (LINQ)
- où la clause

Effectuer des jointures internes

18/03/2020 • 15 minutes to read • Edit Online

Dans le domaine des bases de données relationnelles, une *jointure interne* produit un jeu de résultats dans lequel chaque élément de la première collection apparaît une fois pour chaque élément correspondant dans la deuxième collection. Si un élément de la première collection n'a pas d'élément correspondant, il n'apparaît pas dans le jeu de résultats. La méthode Join, qui est appelée par la clause `join` en C#, implémente une jointure interne.

Cet article explique comment effectuer quatre variations d'une jointure interne :

- Une jointure interne simple qui met en corrélation des éléments de deux sources de données sur la base d'une clé simple.
- Une jointure interne qui met en corrélation des éléments de deux sources de données sur la base d'une clé *composite*. Une clé composite, qui est une clé composée de plusieurs valeurs, permet de mettre en corrélation des éléments sur la base de plusieurs propriétés.
- Une * jointure multiple* dans laquelle les opérations de jointure consécutives sont ajoutées les unes aux autres.
- Une jointure interne qui est implémentée à l'aide d'une jointure groupée.

Exemple – Jointure de clé simple

L'exemple suivant crée deux collections qui contiennent des objets de deux types définis par l'utilisateur, `Person` et `Pet`. La requête utilise la clause `join` dans C# pour faire correspondre les objets `Person` avec les objets `Pet` dont la propriété `Owner` est `Person`. La clause `select` dans C# définit l'apparence des objets résultants. Dans cet exemple, les objets résultants sont des types anonymes qui se composent du prénom du propriétaire et du nom de l'animal domestique.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui
```

Notez que l'objet `Person` dont la propriété `LastName` est « Huff » n'apparaît pas dans le jeu de résultats, car il n'y a pas d'objet `Pet` ayant une propriété `Pet.Owner` égale à `Person`.

Exemple – Jointure de clé composite

Au lieu de mettre en corrélation des éléments sur la base d'une seule propriété, vous pouvez utiliser une clé composite pour comparer des éléments en fonction de plusieurs propriétés. Pour ce faire, spécifiez la fonction de sélection de clé de chaque collection pour retourner un type anonyme qui se compose des propriétés à comparer. Si vous étiquetez les propriétés, elles doivent avoir la même étiquette dans le type anonyme de chaque clé. Les propriétés doivent également apparaître dans le même ordre.

L'exemple suivant utilise une liste d'objets `Employee` et une liste d'objets `Student` pour déterminer quels employés sont également étudiants. Ces deux types ont des propriétés `FirstName` et `LastName` de type String. Les fonctions qui créent les clés de jointure à partir des éléments de chaque liste retournent un type anonyme qui se compose des propriétés `FirstName` et `LastName` de chaque élément. L'opération de jointure effectue une comparaison d'égalité de ces clés composites et retourne les paires d'objets de chaque liste où il y a correspondance entre le prénom et le nom.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
         new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
         new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
         new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Exemple – Jointure multiple

Vous pouvez effectuer une jointure multiple en ajoutant n'importe quel nombre d'opérations de jointure les unes aux autres. Chaque clause `join` dans C# met en corrélation une source de données spécifiée avec les résultats de la jointure précédente.

L'exemple suivant crée trois collections : une liste d'objets `Person`, une liste d'objets `Cat` et une liste d'objets `Dog`.

La première clause `join` dans C# met en corrélation des personnes et des chats sur la base d'un objet `Person` correspondant à `Cat.Owner`. Elle retourne une séquence de types anonymes qui contiennent l'objet `Person` et la propriété `Cat.Name`.

La deuxième clause `join` dans C# met en corrélation les types anonymes retournés par la première jointure avec les objets `Dog` de la liste de chiens fournie, sur la base d'une clé composite constituée de la propriété `Owner` de type `Person` et de la première lettre du nom de l'animal. Elle retourne une séquence de types anonymes qui contiennent les propriétés `Cat.Name` et `Dog.Name` de chaque paire correspondante. Comme il s'agit d'une jointure interne, seuls les objets de la première source de données qui ont une correspondance dans la deuxième source de données sont retournés.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
```

```
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"
{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".
```

Exemple – Jointure interne à l'aide d'une jointure groupée

L'exemple suivant montre comment implémenter une jointure interne en utilisant une jointure groupée.

Dans `query1`, la liste d'objets `Person` est jointe par groupe à la liste d'objets `Pet` sur la base de l'objet `Person` qui correspond à la propriété `Pet.Owner`. La jointure groupée crée une collection de groupes intermédiaires où chaque groupe se compose d'un objet `Person` et d'une séquence d'objets `Pet` correspondants.

En ajoutant une deuxième clause `from` à la requête, cette séquence de séquences est combinée (ou aplatie) dans une séquence plus longue. Le type des éléments de la séquence finale est spécifié par la clause `select`. Dans cet exemple, ce type est un type anonyme qui se compose des propriétés `Person.FirstName` et `Pet.Name` pour chaque paire correspondante.

Le résultat de `query1` est équivalent au jeu de résultats qui aurait été obtenu en utilisant la clause `join` sans la clause `into` pour effectuer une jointure interne. La variable `query2` illustre cette requête équivalente.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists
```

```
    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into gj
                 from subpet in gj
                 select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName = pet.Name };

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures groupées
- Effectuer des jointures externes gauches
- Types anonymes

Effectuer des jointures groupées

02/11/2020 • 7 minutes to read • Edit Online

La jointure groupée est utile pour produire des structures de données hiérarchiques. Elle associe chaque élément de la première collection à un jeu d'éléments corrélés de la deuxième collection.

Par exemple, une classe ou une table de base de données relationnelle nommée `Student` peut contenir deux champs : `Id` et `Name`. Une deuxième classe ou table de base de données relationnelle nommée `Course` peut contenir deux champs : `StudentId` et `CourseTitle`. Une jointure groupée de ces deux sources de données, basée sur les champs `Student.Id` et `Course.StudentId` correspondants, regroupe chaque `Student` avec une collection d'objets `Course` (qui peut être vide).

> **NOTE**
>
> Chaque élément de la première collection apparaît dans le jeu de résultats d'une jointure groupée, même si des éléments corrélés sont trouvés dans la deuxième collection. Si aucun élément corrélé n'est trouvé, la séquence d'éléments corrélés pour cet élément est vide. Le sélecteur de résultats a donc accès à chaque élément de la première collection. Cela n'est pas le cas du sélecteur de résultats dans une jointure non groupée, qui ne peut pas accéder à des éléments de la première collection qui n'ont aucune correspondance dans la deuxième collection.

> **WARNING**
>
> Enumerable.GroupJoin n'a aucun équivalent direct dans les termes de base de données relationnelle traditionnels. Toutefois, cette méthode implémente un sur-ensemble de jointures internes et de jointures externes gauches. Ces deux opérations peuvent être écrites en termes de jointure groupée. Pour plus d'informations, consultez join, opérations et Entity Framework Core, GroupJoin.

Le premier exemple de cet article montre comment effectuer une jointure groupée. Le deuxième exemple montre comment utiliser une jointure groupée pour créer des éléments XML.

Exemple – Jointure groupée

L'exemple suivant effectue une jointure groupée d'objets de types `Person` et `Pet` où `Person` correspond à la propriété `Pet.Owner`. Contrairement à une jointure non groupée qui produirait une paire d'éléments pour chaque correspondance, la jointure groupée produit un seul objet résultant pour chaque élément de la première collection, qui est un objet `Person` dans cet exemple. Les éléments correspondants de la deuxième collection, qui sont des objets `Pet` dans cet exemple, sont regroupés dans une collection. Enfin, le sélecteur de résultats crée un type anonyme pour chaque correspondance qui se compose de `Person.FirstName` et d'une collection d'objets `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

Exemple – Jointure groupée pour créer des éléments XLM

Les jointures groupées sont appropriées pour créer des éléments XML à l'aide de LINQ to XML. L'exemple suivant est similaire à l'exemple précédent, sauf qu'au lieu de créer des types anonymes, le sélecteur de résultats crée des

éléments XML qui représentent les objets joints.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures internes
- Effectuer des jointures externes gauches
- Types anonymes

Effectuer des jointures externes gauches

18/03/2020 • 3 minutes to read • Edit Online

Une jointure externe gauche est une jointure dans laquelle chaque élément de la première collection est retourné, qu'elle ait ou non des éléments corrélés dans la deuxième collection. Vous pouvez utiliser LINQ pour effectuer une jointure externe gauche en appelant la méthode DefaultIfEmpty sur les résultats d'une jointure groupée.

Exemple

L'exemple suivant montre comment utiliser la méthode DefaultIfEmpty sur les résultats d'une jointure groupée pour effectuer une jointure externe gauche.

Pour créer une jointure externe gauche entre deux collections, la première étape consiste à effectuer une jointure interne à l'aide d'une jointure groupée. (Voir Perform inner se joint pour une explication de ce processus.) Dans cet exemple, `Person` la liste des objets est `Pet` jointe à `Person` la `Pet.Owner` liste des objets sur la base d'un objet qui correspond .

La deuxième étape consiste à inclure tous les éléments de la première collection (celle de gauche) dans le jeu de résultats, y compris les éléments sans correspondance dans la collection de droite. Pour cela, appelez DefaultIfEmpty sur chaque séquence d'éléments correspondants de la jointure groupée. Dans cet exemple, la méthode DefaultIfEmpty est appelée sur chaque séquence d'objets `Pet` correspondants. La méthode retourne une collection qui contient uniquement une valeur par défaut si la séquence des objets `Pet` correspondants à un objet `Person` est vide, ce qui garantit que chaque objet `Person` est représenté dans la collection de résultats.

> **NOTE**
>
> La valeur par défaut pour un type référence est `null`. L'exemple recherche donc une référence null avant d'accéder à chaque élément de chaque collection `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures intérieures
- Effectuer des jointures groupées
- Types anonymes

Classer les résultats d'une clause join

18/03/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment classer les résultats d'une opération de jointure. Notez que le classement est effectué après la jointure. Vous pouvez utiliser une clause `orderby` avec une ou plusieurs séquences sources avant la jointure, mais cette pratique est généralement déconseillée. Certains fournisseurs LINQ ne conservent pas ce classement après la jointure.

Exemple

Cette requête crée une jointure groupée, puis trie les groupes en fonction de l'élément de catégorie, qui est encore dans la portée. À l'intérieur de l'initialiseur de type anonyme, une sous-requête classe tous les éléments correspondants de la séquence de produits.

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
 {
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
 };

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
 };
        #endregion
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();

            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
```

```
            Console.ReadKey();
        }

        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.CategoryID into prodGroup
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine($"  {prodItem.Name,-10} {prodItem.CategoryID}");
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

Voir aussi

- Requête intégrée linguistique (LINQ)
- orderby, clause
- clause d'adhésion

Effectuer des jointures à l'aide de clés composites

18/03/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment effectuer des opérations de jointure où vous voulez utiliser plusieurs clés pour définir une correspondance. Ceci s'effectue à l'aide d'une clé composite. Vous créez une clé composite en tant que type anonyme ou typé nommé avec les valeurs que vous voulez comparer. Si la variable de requête doit être passée au-delà des limites de la méthode, utilisez un type nommé qui substitue Equals et GetHashCode pour la clé. Les noms des propriétés et l'ordre dans lequel elles se trouvent doivent être identiques dans chaque clé.

Exemple

L'exemple suivant montre comment utiliser une clé composite pour joindre les données de trois tables :

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

L'inférence du type sur les clés composites dépend des noms des propriétés dans les clés et l'ordre dans lequel elles apparaissent. Si les propriétés dans les séquences sources n'ont pas les mêmes noms, vous devez affecter de nouveaux noms dans les clés. Par exemple, si la table `Orders` et la table `OrderDetails` utilisent chacune des noms différents pour leurs colonnes, vous pouvez créer des clés composites en affectant des noms identiques dans les types anonymes :

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Vous pouvez aussi utiliser des clés composites dans une clause `group`.

Voir aussi

- Requête intégrée linguistique (LINQ)
- clause d'adhésion
- group, clause

Effectuer des opérations de jointure personnalisées

18/03/2020 • 7 minutes to read • Edit Online

Cet exemple montre comment effectuer des opérations de jointure qui ne sont pas possibles avec la clause `join`. Dans une expression de requête, la clause `join` est limitée à et optimisée pour les équijointures, qui sont de loin le type le plus courant d'opération de jointure. Quand vous effectuez une équijointure, l'utilisation de la clause `join` offre généralement les meilleures performances.

Toutefois, la clause `join` ne peut pas être utilisée dans les cas suivants :

- Quand la jointure est basée sur une expression de prédicat d'inégalité (une non-équijointure).
- Quand la jointure est basée sur plusieurs expressions de prédicat d'égalité ou d'inégalité.
- Quand vous devez introduire une variable de portée temporaire pour la séquence de droite (interne) avant l'opération de jointure.

Pour effectuer des jointures qui ne sont pas des équijointures, vous pouvez utiliser plusieurs clauses `from` pour introduire chaque source de données indépendamment. Vous appliquez ensuite une expression de prédicat dans une clause `where` à la variable de portée pour chaque source. L'expression peut également prendre la forme d'un appel de méthode.

> **NOTE**
> Ne confondez pas ce type d'opération de jointure personnalisée avec l'utilisation de plusieurs clauses `from` permettant d'accéder aux collections internes. Pour plus d'informations, voir clause d'adhésion.

Exemple

Dans l'exemple suivant, la première méthode présente une jointure croisée simple. Les jointures croisées doivent être utilisées avec précaution, car elles peuvent produire des jeux de résultats très volumineux. Toutefois, elles peuvent être utiles dans certains scénarios pour créer des séquences sources sur lesquelles exécuter des requêtes supplémentaires.

La deuxième méthode produit une séquence de tous les produits dont l'ID de catégorie est répertorié dans la liste des catégories à gauche. Notez l'utilisation de la clause `let` et de la méthode `Contains` pour créer un tableau temporaire. Il est également possible de créer le tableau avant la requête et de supprimer la première clause `from`.

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }
```

```
        // Specify the first data source.
        List<Category> categories = new List<Category>()
        {
            new Category(){Name="Beverages", ID=001},
            new Category(){ Name="Condiments", ID=002},
            new Category(){ Name="Vegetables", ID=003},
        };

        // Specify the second data source.
        List<Product> products = new List<Product>()
        {
            new Product{Name="Tea",  CategoryID=001},
            new Product{Name="Mustard", CategoryID=002},
            new Product{Name="Pickles", CategoryID=002},
            new Product{Name="Carrots", CategoryID=003},
            new Product{Name="Bok Choy", CategoryID=003},
            new Product{Name="Peaches", CategoryID=005},
            new Product{Name="Melons", CategoryID=005},
            new Product{Name="Ice Cream", CategoryID=007},
            new Product{Name="Mackerel", CategoryID=012},
        };
        #endregion

        static void Main()
        {
            CustomJoins app = new CustomJoins();
            app.CrossJoin();
            app.NonEquijoin();

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }

        void CrossJoin()
        {
            var crossJoinQuery =
                from c in categories
                from p in products
                select new { c.ID, p.Name };

            Console.WriteLine("Cross Join Query:");
            foreach (var v in crossJoinQuery)
            {
                Console.WriteLine($"{v.ID,-5}{v.Name}");
            }
        }

        void NonEquijoin()
        {
            var nonEquijoinQuery =
                from p in products
                let catIds = from c in categories
                             select c.ID
                where catIds.Contains(p.CategoryID) == true
                select new { Product = p.Name, CategoryID = p.CategoryID };

            Console.WriteLine("Non-equijoin query:");
            foreach (var v in nonEquijoinQuery)
            {
                Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
            }
        }
    }
    /* Output:
Cross Join Query:
1    Tea
1    Mustard
1    Pickles
```

```
1    Carrots
1    Bok Choy
1    Peaches
1    Melons
1    Ice Cream
1    Mackerel
2    Tea
2    Mustard
2    Pickles
2    Carrots
2    Bok Choy
2    Peaches
2    Melons
2    Ice Cream
2    Mackerel
3    Tea
3    Mustard
3    Pickles
3    Carrots
3    Bok Choy
3    Peaches
3    Melons
3    Ice Cream
3    Mackerel
Non-equijoin query:
1    Tea
2    Mustard
2    Pickles
3    Carrots
3    Bok Choy
Press any key to exit.
     */
```

Exemple

Dans l'exemple suivant, la requête doit joindre deux séquences sur la base des clés correspondantes qui, pour la séquence interne (à droite), ne peuvent pas être obtenues avant la clause de jointure elle-même. Si cette jointure a été effectuée avec une clause `join`, la méthode `Split` doit être appelée pour chaque élément. L'utilisation de plusieurs clauses `from` permet d'éviter à la requête la surcharge inhérente à la répétition de l'appel de méthode. Toutefois, comme la clause `join` est optimisée, dans ce cas particulier, son utilisation peut s'avérer plus rapide que d'utiliser plusieurs clauses `from`. Les résultats dépendront principalement de la surcharge liée à l'appel de la méthode.

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
```

```
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

Voir aussi

- Requête intégrée linguistique (LINQ)
- clause d'adhésion
- Classer les résultats d'une clause join

Gérer des valeurs Null dans des expressions de requête

27/03/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment gérer d'éventuelles valeurs Null dans des collections sources. Une collection d'objets telle qu'un IEnumerable<T> peut contenir des éléments dont la valeur est null. Si une collection source est null ou contient un élément dont la valeur est null et que votre requête ne gère pas les valeurs null, une NullReferenceException est levée quand vous exécutez la requête.

Exemple

Vous pouvez écrire du code en prévention pour éviter une exception de référence Null, comme indiqué dans l'exemple suivant :

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

Dans l'exemple précédent, la clause `where` exclut tous les éléments Null de la séquence de catégories. Cette technique est indépendante de la vérification de valeur Null de la clause join. Dans cet exemple, l'expression conditionnelle avec une valeur Null fonctionne, car `Products.CategoryID` est de type `int?`, qui est le raccourci de `Nullable<int>`.

Exemple

Dans une clause de jointure, si une seule des clés de comparaison est un type de valeur nul, vous pouvez jeter l'autre à un type de valeur nul dans l'expression de la requête. Dans l'exemple suivant, supposons que `EmployeeID` soit une colonne qui contienne des valeurs de type `int?` :

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

Voir aussi

- Nullable<T>
- Requête intégrée linguistique (LINQ)
- types valeur Nullable

Gérer des exceptions dans des expressions de requête

18/03/2020 • 3 minutes to read • Edit Online

Dans le contexte d'une expression de requête, vous pouvez appeler n'importe quelle méthode. Toutefois, nous vous recommandons d'éviter d'appeler une méthode dans une expression de requête susceptible de créer un effet secondaire, tel que la modification du contenu de la source de données ou la levée d'une exception. Cet exemple montre comment éviter la levée d'exceptions lorsque vous appelez des méthodes dans une expression de requête, en respectant les directives générales .NET relatives à la gestion des exceptions. Selon ces directives, il est acceptable d'intercepter une exception spécifique si vous comprenez pourquoi elle est levée dans un contexte donné. Pour plus d'informations, voir Meilleures pratiques pour les exceptions.

Le dernier exemple montre comment gérer les cas où vous devez lever une exception pendant l'exécution d'une requête.

Exemple

L'exemple suivant montre comment déplacer du code de gestion des exceptions en dehors d'une expression de requête. Ceci est possible uniquement lorsque la méthode ne dépend pas des variables locales de la requête.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Exemple

Dans certains cas, la meilleure réponse à la levée d'une exception à l'intérieur d'une requête consiste à arrêter immédiatement l'exécution de la requête. L'exemple suivant montre comment gérer les exceptions pouvant être levées dans le corps d'une requête. Supposons que `SomeMethodThatMightThrow` puisse provoquer la levée d'une exception qui nécessite l'arrêt de l'exécution de la requête.

Notez que le bloc `try` englobe la boucle `foreach` et non la requête. Ceci s'explique par le fait que c'est au niveau de la boucle `foreach` que la requête est exécutée. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Voir aussi

- Requête intégrée linguistique (LINQ)

Programmation asynchrone

16/12/2020 • 20 minutes to read • Edit Online

Si vous avez des besoins liés aux e/s (par exemple, la demande de données à un réseau, l'accès à une base de données ou la lecture et l'écriture dans un système de fichiers), vous pouvez utiliser la programmation asynchrone. L'écriture de code asynchrone est également indiquée si votre code utilise le processeur de manière intensive, notamment pour effectuer un calcul complexe.

C# possède un modèle de programmation asynchrone au niveau du langage, qui permet d'écrire facilement du code asynchrone sans avoir à jongler avec les rappels ni à se conformer à une bibliothèque qui prend en charge l'asynchronie. Il suit ce que l'on appelle le modèle asynchrone basé sur des tâches (TAP).

Vue d'ensemble du modèle asynchrone

La programmation asynchrone est basée sur les objets `Task` et `Task<T>`, qui modélisent les opérations asynchrones. Ces objets sont exposés à l'aide des mots clés `async` et `await`. Dans la plupart des cas, le modèle est assez simple :

- Pour le code lié aux e/s, vous attendez une opération qui retourne un `Task` ou `Task<T>` à l'intérieur d'une `async` méthode.
- Pour le code lié au processeur, vous attendez une opération démarrée sur un thread d'arrière-plan avec la Task.Run méthode.

Le mot clé `await` trouve ici toute son utilité. Il cède le contrôle à l'appelant de la méthode qui a effectué l'opération `await`. Au final, c'est ce qui rend une interface utilisateur réactive ou un service élastique. Bien qu' il existe des façons d'aborder du code asynchrone autre que `async` et `await` , cet article se concentre sur les constructions de niveau de langage.

Exemple de liaison d'e/s : Télécharger des données à partir d'un service Web

Vous devrez peut-être télécharger des données à partir d'un service Web lorsque vous appuyez sur un bouton mais que vous ne souhaitez pas bloquer le thread d'interface utilisateur. Il peut être accompli de la façon suivante :

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

Le code exprime l'intention (en téléchargeant les données de façon asynchrone) sans être coincés dans l'interaction avec les `Task` objets.

Exemple de processeur dépendant : effectuer un calcul pour un jeu

Supposons que vous développez un jeu pour mobile dans lequel l'appui sur un bouton peut causer des dommages à de nombreux ennemis à l'écran. Le calcul des dommages infligés peut nécessiter beaucoup de ressources. Si ce calcul est effectué sur le thread d'interface utilisateur, le jeu risque d'être considérablement

ralenti pendant la durée du calcul.

La meilleure façon de gérer cela est de démarrer un thread d'arrière-plan, qui effectue le travail à l'aide de `Task.Run` , et d'attendre son résultat à l'aide de `await` . Cela permet à l'interface utilisateur de paraître lisse au fur et à mesure que le travail est effectué.

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Ce code exprime clairement l'objectif de l'événement Click du bouton, il ne nécessite pas de gérer un thread d'arrière-plan manuellement, et il le fait de façon non bloquante.

Les dessous du code

Il y a de nombreux éléments mobiles dans lesquels des opérations asynchrones sont concernées. Si vous êtes curieux de savoir ce qui se passe sous les couvertures de `Task` et `Task<T>` , consultez l'article Async in-Depth pour plus d'informations.

Du côté du C#, le compilateur transforme votre code en une machine à États qui effectue le suivi d'opérations telles que le traitement de l'exécution quand un `await` est atteint et la reprise de l'exécution lorsqu'une tâche en arrière-plan est terminée.

D'un point de vue théorique, il s'agit d'une implémentation du modèle de promesses d'asynchronisme.

Éléments clés à comprendre

- Le code asynchrone peut être utilisé pour du code utilisant les E/S ou le processeur de manière intensive, mais il est utilisé de manière différente dans chaque scénario.
- Le code asynchrone utilise les objets `Task<T>` et `Task`, qui sont des constructions servant à modéliser le travail effectué en arrière-plan.
- Le mot clé `async` définit une méthode comme asynchrone, ce qui vous permet d'utiliser le mot clé `await` dans le corps de la méthode.
- Quand le mot clé `await` est utilisé, il suspend la méthode d'appel et cède le contrôle à son appelant jusqu'à ce que la tâche awaited soit terminée.
- Le mot clé `await` peut uniquement être utilisé dans une méthode asynchrone.

Reconnaître le travail lié au processeur et à l'e/s

Les deux premiers exemples de ce guide ont montré comment utiliser `async` et pour le travail lié aux `await` e/s et au processeur. C'est la clé que vous pouvez identifier lorsqu'un travail que vous devez effectuer est lié aux e/s ou à l'UC, car cela peut affecter les performances de votre code et peut potentiellement entraîner une utilisation erronée de certaines constructions.

Voici deux questions à vous poser avant d'écrire du code :

1. Votre code doit-il « attendre » quelque chose, par exemple des données d'une base de données ?

   Si la réponse est « oui », le travail **utilise les E/S de manière intensive**.

2. Votre code effectuera-t-il un calcul coûteux ?

   Si la réponse est « oui », le travail **utilise le processeur de manière intensive**.

Si le travail à faire **utilise les E/S de manière intensive**, utilisez `async` et `await` *sans* `Task.Run`. Vous *ne devez pas* utiliser la bibliothèque parallèle de tâches. Cette raison est décrite en détail dans Async.

Si le travail que vous avez est lié à l' **UC** et que vous vous souciez de la réactivité, utilisez `async` et `await` , mais générez le travail sur un autre thread *avec* `Task.Run` . Si le travail est approprié pour la concurrence et le parallélisme, envisagez également d'utiliser la bibliothèque parallèle de tâches.

De plus, vous devez toujours mesurer les performances d'exécution de votre code. Par exemple, vous constaterez peut-être que le coût d'un travail utilisant le processeur de manière intensive n'est pas si élevé que cela par rapport à la surcharge des changements de contexte induits par le multithreading. Chaque solution ayant ses compromis, choisissez le meilleur compromis pour votre scénario.

Plus d'exemples

Les exemples suivants montrent diverses façons d'écrire du code asynchrone dans C#. Ils correspondent à plusieurs scénarios différents que vous êtes susceptible de rencontrer.

Extraire des données à partir d'un réseau

Cet extrait de code télécharge le code HTML à partir de la page d'accueil de https://dotnetfoundation.org et compte le nombre de fois que la chaîne « .net » se produit dans le code html. Elle utilise ASP.NET pour définir une méthode de contrôleur d'API Web qui effectue cette tâche et retourne le nombre.

> **NOTE**
> Si vous prévoyez d'effectuer une analyse HTML dans le code de production, n'utilisez pas d'expressions régulières. Utilisez plutôt une bibliothèque d'analyse.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Voici le même scénario écrit pour une application Windows universelle, qui effectue la même tâche quand l'utilisateur appuie sur un bouton :

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Attendre la fin de plusieurs tâches

Vous pouvez avoir un scénario qui nécessite de récupérer plusieurs éléments de données simultanément. L' `Task` API contient deux méthodes, Task.WhenAll et Task.WhenAny , qui vous permettent d'écrire du code asynchrone qui exécute une attente non bloquante sur plusieurs travaux en arrière-plan.

Cet exemple vous montre comment récupérer des données `User` pour plusieurs `userId`.

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Voici une autre façon de l'écrire de façon plus succincte, à l'aide de LINQ :

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

Bien qu'il s'agisse d'un code moins important, soyez prudent lors de la combinaison de LINQ et du code asynchrone. Du fait que LINQ utilise l'exécution différée, les appels asynchrones ne sont pas effectués immédiatement comme c'est le cas dans une boucle `foreach`, sauf si vous forcez l'itération de la séquence générée avec un appel à `.ToList()` ou `.ToArray()`.

Informations et conseils importants

Avec la programmation asynchrone, il existe des détails à garder à l'esprit qui peuvent empêcher un comportement inattendu.

- **`async` Les méthodes doivent contenir un mot clé `await` dans leur corps pour pouvoir être suspendues.**

  Il ne faut pas oublier ce point. Si `await` n'est pas utilisé dans le corps d'une `async` méthode, le compilateur C# génère un avertissement, mais le code se compile et s'exécute comme s'il s'agissait d'une méthode normale. Cela est incroyablement inefficace, car l'ordinateur d'État généré par le compilateur C# pour la méthode Async n'accomplit rien.

- **Vous devez ajouter « Async » comme suffixe de chaque nom de méthode Async que vous écrivez.**

Il s'agit de la Convention utilisée dans .NET pour différencier plus facilement les méthodes synchrones et asynchrones. Certaines méthodes qui ne sont pas explicitement appelées par votre code (telles que les gestionnaires d'événements ou les méthodes de contrôleur Web) ne s'appliquent pas nécessairement. Étant donné qu'elles ne sont pas explicitement appelées par votre code, il n'est pas aussi important d'être explicite quant à leur nom.

- **`async void` ne doit être utilisé que pour les gestionnaires d'événements.**

L'utilisation de `async void` est le seul moyen de permettre le fonctionnement des gestionnaires d'événements asynchrones, car les événements n'ont pas de types de retour (et ne peuvent donc pas utiliser les objets `Task` et `Task<T>`). Toute autre utilisation de la méthode `async void` ne suit pas le modèle TAP et peut être difficile à implémenter, comme expliqué ci-après :

- Les exceptions levées dans une `async void` méthode ne peuvent pas être interceptées en dehors de cette méthode.
- `async void` les méthodes sont difficiles à tester.
- `async void` les méthodes peuvent provoquer des effets secondaires incorrects si l'appelant ne s'attend pas à ce qu'ils soient asynchrones.
- **Définissez le thread avec précaution si vous utilisez des expressions lambda asynchrones dans des expressions LINQ.**

Les expressions lambda dans LINQ utilisent l'exécution différée, ce qui signifie que le code peut finir à s'exécuter à un moment où vous ne l'attendiez pas. L'introduction de tâches bloquantes dans ce code peut facilement provoquer un interblocage si le code n'est pas écrit correctement. De plus, l'imbrication de code asynchrone peut rendre la logique d'exécution du code plus compliquée. Async et LINQ sont puissants, mais doivent être utilisés ensemble aussi bien que possible.

- **Écrivez du code qui attend certaines tâches de façon non bloquante.**

Le fait de bloquer le thread actuel comme un moyen d'attendre qu'un `Task` se termine peut entraîner des interblocages et bloquer les threads de contexte et peut nécessiter une gestion des erreurs plus complexe. Le tableau suivant fournit des conseils sur la façon de traiter l'attente des tâches de façon non bloquante :

ÉLÉMENT À UTILISER...	AU LIEU DE...	LORSQUE VOUS SOUHAITEZ EFFECTUER CETTE OPÉRATION...
`await`	`Task.Wait` ou `Task.Result`	Extraire le résultat d'une tâche en arrière-plan
`await Task.WhenAny`	`Task.WaitAny`	Attendre la fin d'une tâche
`await Task.WhenAll`	`Task.WaitAll`	Attendre la fin de toutes les tâches
`await Task.Delay`	`Thread.Sleep`	Attendre pendant une période

- **Envisagez d'utiliser `ValueTask` dans** la mesure du possible

Le retour d'un objet `Task` à partir de méthodes async peut introduire des goulots d'étranglement au niveau des performances dans certains chemins. `Task` est un type référence. Si vous l'utilisez, vous allouez donc un objet. Dans les cas où une méthode déclarée avec le `async` modificateur retourne un résultat mis en cache ou se termine de façon synchrone, les allocations supplémentaires peuvent devenir un coût de temps significatif dans les sections de code critiques pour les performances. Cela peut devenir coûteux si ces allocations se produisent dans des boucles serrées. Pour plus d'informations, consultez types de retour asynchrones généralisés.

- **Envisagez d'utiliser `ConfigureAwait(false)`**

Une question courante est « quand dois-je utiliser la Task.ConfigureAwait(Boolean) méthode ? ». La méthode permet à une `Task` instance de configurer son await. Il s'agit d'un élément important à prendre en compte et sa définition de manière incorrecte peut avoir une incidence sur les performances et même des blocages. Pour plus d'informations sur `ConfigureAwait` , consultez le Forum aux questions ConfigureAwait.

- **Limitez l'écriture de code avec état.**

Ne dépendez pas de l'état des objets globaux ou de l'exécution de certaines méthodes. Le code doit uniquement dépendre des valeurs de retour des méthodes. Pourquoi ?

- La logique du code sera plus facile à comprendre.
- Le code sera plus facile à tester.
- Combiner du code asynchrone et du code synchrone est beaucoup plus simple.
- Les concurrences critiques peuvent généralement être évitées.
- Rendre le code dépendant des valeurs de retour facilite la coordination du code asynchrone.
- En prime, le code fonctionne parfaitement avec l'injection de dépendances.

L'objectif recommandé est d'atteindre une transparence référentielle complète ou quasi-complète dans votre code. Votre base de code sera alors prédictible, testable et facile à gérer.

Autres ressources

- L'article Async en détail fournit des informations supplémentaires sur le fonctionnement des tâches.
- Programmation asynchrone avec Async et await (C#)
- Les vidéos Six Essential Tips for Async de Lucian Wischik sont une ressource très utile pour la programmation asynchrone.

Critères spéciaux

02/11/2020 • 24 minutes to read • Edit Online

Les modèles vérifient qu'une valeur a une certaine *forme* et peuvent *extraire* des informations de la valeur quand celle-ci a la forme correspondante. Les critères spéciaux offrent une syntaxe plus concise pour les algorithmes que vous utilisez déjà aujourd'hui. Vous pouvez déjà créer des algorithmes de critères spéciaux à l'aide de la syntaxe existante. Pour cela, vous écrivez des instructions `if` ou `switch` qui testent une valeur. Ensuite, quand ces instructions correspondent, vous extrayez et utilisez les informations de cette valeur. Les nouveaux éléments de syntaxe étendent les instructions que vous connaissez déjà : `is` et `switch`. Ces nouvelles extensions permettent à la fois de tester une valeur et d'extraire les informations de cette valeur.

Dans cet article, nous allons examiner la nouvelle syntaxe pour voir en quoi elle rend le code plus lisible et plus concis. Les critères spéciaux permettent d'utiliser des idiomes dans lesquels le code et les données sont séparés, contrairement aux conceptions orientées objet où les données et les méthodes qui les manipulent sont étroitement couplées.

Pour illustrer ces nouveaux idiomes, nous allons manipuler des structures représentant des formes géométriques à l'aide d'instructions de critères spéciaux. Vous savez probablement déjà concevoir des hiérarchies de classes et créer des méthodes virtuelles et méthodes substituées pour personnaliser le comportement d'un objet en fonction de son type à l'exécution.

Ces techniques ne sont pas possibles avec des données qui ne sont pas structurées dans une hiérarchie de classes. Quand les données et les méthodes sont séparées, vous avez besoin d'autres outils. Les nouvelles constructions de *critères spéciaux* utilisent une syntaxe simplifiée pour examiner les données et manipuler le flux de contrôle en fonction de la condition de ces données. Vous écrivez déjà des instructions `if` et `switch` qui testent la valeur d'une variable. Vous écrivez déjà des instructions `is` qui testent le type d'une variable. Les *critères spéciaux* ajoutent de nouvelles fonctionnalités à ces instructions.

Dans cet article, vous allez créer une méthode qui calcule la surface de différentes formes géométriques. Pour cela, vous n'allez pas utiliser de techniques orientées objet, ni créer une hiérarchie de classes pour les différentes formes. Vous allez utiliser des *critères spéciaux* à la place. Tout au long de cet exemple, comparez ce code avec le même code structuré comme une hiérarchie d'objets. Quand les données à interroger et à manipuler ne constituent pas une hiérarchie de classes, les critères spéciaux permettent de réaliser des conceptions élégantes.

Au lieu de commencer avec une définition de forme abstraite et d'ajouter ensuite différentes classes de formes spécifiques, écrivez d'abord les définitions de données simples pour chacune des formes géométriques :

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}
public struct Rectangle
{
    public double Length { get; }
    public double Height { get; }

    public Rectangle(double length, double height)
    {
        Length = length;
        Height = height;
    }
}
public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}
```

À partir de ces structures, écrivez une méthode qui calcule la surface d'une forme.

Expression du modèle de type `is`

Avant C# 7.0, vous deviez tester chaque type dans une série d'instructions `if` et `is` :

```
public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

Le code ci-dessus est une expression standard du *modèle de type* : vous testez une variable pour déterminer son type et vous effectuez une action différente en fonction de ce type.

Vous pouvez simplifier ce code en ajoutant des extensions à l'expression `is` pour assigner une variable si le test réussit :

```
public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

Dans cette version mise à jour, l'expression `is` teste la variable et l'assigne à une nouvelle variable du type approprié. Remarquez également que cette version inclut le type `Rectangle`, qui est un `struct`. La nouvelle expression `is` peut être utilisée avec des types valeur et des types référence.

Les règles de langage des expressions de critères spéciaux vous aident à éviter une utilisation incorrecte des résultats d'une expression de correspondance. Dans l'exemple ci-dessus, les variables `s`, `c` et `r` sont uniquement dans la portée et assignées de manière définitive quand les expressions de critères spéciaux respectives retournent le résultat `true`. Si vous essayez d'utiliser l'une de ces variables à un autre emplacement, votre code génère des erreurs de compilateur.

Examinons ces deux règles en détail, en commençant par la portée. La variable `c` est dans la portée uniquement dans la branche `else` de la première instruction `if`. La variable `s` est dans la portée dans la méthode `ComputeAreaModernIs`. Cela est dû au fait que chaque branche d'une instruction `if` établit une portée distincte pour les variables, ce que ne fait pas l'instruction `if` proprement dite. Cela signifie que les variables déclarées dans l' `if` instruction se trouvent dans la même portée que l' `if` instruction (dans ce cas, la méthode). Ce comportement n'est pas propre aux critères spéciaux. C'est le comportement défini pour les portées de variables et pour les instructions `if` et `else`.

Les variables `c` et `s` sont assignées quand les instructions `if` respectives ont la valeur true, selon la règle d'assignation définitive quand le résultat est true.

TIP

Les exemples de cette rubrique utilisent la construction recommandée où une expression de critères spéciaux `is` assigne de manière définitive la variable de correspondance dans la branche `true` de l'instruction `if`. Vous pourriez inverser la logique en écrivant `if (!(shape is Square s))`. La variable `s` serait alors assignée de manière définitive uniquement dans la branche `false`. Cette pratique est possible dans C#, mais elle est déconseillée, car elle repose sur une logique plus compliquée à suivre.

Ces règles limitent le risque d'accéder accidentellement au résultat d'une expression de critères spéciaux dont le test n'a pas réussi.

Utilisation des instructions de critères spéciaux `switch`

Au fil du temps, vous aurez peut-être besoin de prendre en charge d'autres types de formes. Si vous avez de plus en plus de conditions à tester, utiliser les expressions de critères spéciaux `is` peut devenir compliqué. En plus de nécessiter des instructions `if` pour chaque type à vérifier, les expressions `is` permettent uniquement de vérifier si l'entrée correspond à un type unique. Dans ce cas, les expressions de critères spéciaux `switch` sont plus adaptées.

L'instruction `switch` standard était une expression de modèle, qui prenait en charge le modèle de constante. Vous pouviez comparer une variable à n'importe quelle constante utilisée dans une instruction `case` :

```
public static string GenerateMessage(params string[] parts)
{
    switch (parts.Length)
    {
        case 0:
            return "No elements to the input";
        case 1:
            return $"One element: {parts[0]}";
        case 2:
            return $"Two elements: {parts[0]}, {parts[1]}";
        default:
            return $"Many elements. Too many to write";
    }
}
```

Le seul modèle pris en charge par l'instruction `switch` était le modèle de constante. De plus, il était limité aux types numériques et au type `string`. Ces restrictions ayant été supprimées, vous pouvez maintenant écrire une instruction `switch` en utilisant le modèle de type :

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

L'instruction de critères spéciaux `switch` utilise une syntaxe qui est familière aux développeurs ayant utilisé l'instruction `switch` de style C standard. Chaque instruction `case` est évaluée, et le code en-dessous de la condition qui correspond à la variable d'entrée est exécuté. L'exécution du code ne continue pas d'une expression case à la suivante, car la syntaxe de l'instruction `case` impose que chaque `case` se termine par `break`, `return` ou `goto`.

> **NOTE**
> Les instructions `goto` pour accéder à une autre étiquette sont valides uniquement pour le modèle de constante (l'instruction switch standard).

Il y a de nouvelles règles importantes qui régissent l'instruction `switch`. Les restrictions sur le type de la variable dans l'expression `switch` ont été supprimées. Vous pouvez maintenant utiliser tous les types, comme le type `object` dans cet exemple. Les expressions case ne sont plus limitées à des valeurs de constante. Avec la suppression de cette limitation, la réorganisation des sections `switch` peut changer le comportement d'un programme.

Quand les expressions étaient limitées à des valeurs de constante, une seule étiquette `case` pouvait correspondre à la valeur de l'expression `switch`. La conséquence de cette règle, associée à la règle selon laquelle une section `switch` ne devait pas passer à la section suivante, était que les sections `switch` pouvaient être réorganisées dans n'importe quel ordre sans que cela change le comportement du programme. Maintenant, avec les expressions `switch` généralisées, l'ordre de chaque section est important. Les expressions `switch` sont évaluées dans l'ordre textuel. L'exécution passe à la première étiquette `switch` qui correspond à l'expression `switch`.
Le cas `default` n'est exécuté que si aucune autre étiquette case ne correspond. L'expression case `default` est évaluée en dernier, quel que soit son ordre textuel. S'il n'existe aucun cas `default` ni aucune correspondance avec les autres instructions `case`, l'exécution passe à l'instruction suivant l'instruction `switch`. Aucun code d'étiquettes `case` n'est exécuté.

Clauses `when` dans les expressions `case`

Vous pouvez créer des expressions case spéciales pour les formes de surface 0 en ajoutant une clause `when` dans l'étiquette `case`. Un carré avec une longueur de côté de 0, ou un cercle de rayon 0, a une surface égale à 0. Vous spécifiez cette condition en ajoutant une clause `when` dans l'étiquette `case` :

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Cette modification illustre quelques points importants relatifs à la nouvelle syntaxe. Tout d'abord, plusieurs

étiquettes `case` peuvent être appliquées à une même section `switch`. Le bloc d'instructions est exécuté quand l'une de ces étiquettes est `true`. Dans ce cas, si l'expression `switch` est un cercle ou un carré de surface 0, la méthode retourne la constante 0.

Cet exemple introduit deux variables différentes dans les deux étiquettes `case` du premier bloc `switch`. Notez que les instructions de ce bloc `switch` n'utilisent ni la variable `c` (pour le cercle), ni la variable `s` (pour le carré). Aucune de ces variables n'est assignée de manière définitive dans ce bloc `switch`. Si l'une de ces étiquettes case correspond, cela signifie qu'une des variables a été assignée. Toutefois, il est impossible de déterminer *laquelle* a été assignée au moment de la compilation, car les deux cas peuvent correspondre à l'exécution. C'est pourquoi, la plupart du temps, quand on utilise plusieurs étiquettes `case` dans le même bloc, on n'introduit pas de nouvelle variable dans l'instruction `case` ou on ne l'utilise que dans la clause `when`.

Après avoir ajouté ces formes de surface 0, ajoutez deux autres types de formes, un rectangle et un triangle :

```
public static double ComputeArea_Version4(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Avec cet ensemble de modifications, vous avez ajouté des étiquettes `case` pour l'expression case dégénérée, ainsi que des étiquettes et des blocs pour chacune des nouvelles formes.

Enfin, vous pouvez ajouter un cas `null` pour que l'argument ne soit pas `null` :

```
public static double ComputeArea_Version5(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        case null:
            throw new ArgumentNullException(paramName: nameof(shape), message: "Shape must not be null");
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Le comportement spécial pour le `null` modèle est intéressant, car la constante du `null` modèle n'a pas de type mais peut être convertie en un type référence ou un type valeur Nullable. Plutôt que de convertir une valeur `null` en un type quelconque, le langage définit qu'une valeur `null` ne correspond à aucun modèle de type, quel que soit le type de la variable au moment de la compilation. En raison de ce comportement, le nouveau modèle de type basé sur `switch` est cohérent par rapport à l'instruction `is` : les instructions `is` retournent toujours `false` quand la valeur vérifiée est `null`. Il est aussi plus simple : une fois le type vérifié, aucun autre contrôle de valeur Null n'est nécessaire. On constate en effet qu'aucune vérification de ce type ne figure dans les blocs de cas des exemples ci-dessus : ces contrôles ne sont pas requis dans la mesure où la correspondance avec le modèle de type garantit une valeur non Null.

Déclarations `var` dans les expressions `case`

L'introduction de `var` comme l'une des expressions de correspondance fournit de nouvelles règles pour les critères spéciaux.

La première règle est que la déclaration `var` suit les règles standard d'inférence de type, à savoir que le type est supposé être le type statique de l'expression switch. Avec cette règle, il y a toujours correspondance du type.

La deuxième règle est qu'une déclaration `var` n'a pas le contrôle de valeur Null que comportent d'autres expressions de modèle de type. La variable peut donc être Null, et une vérification de valeur Null est alors nécessaire.

Ces deux règles signifient que, dans de nombreux cas, une déclaration `var` dans une expression `case` remplit les mêmes conditions qu'une expression `default`. Étant donné qu'une expression case non définie par défaut est toujours préférée à l'expression case `default`, l'expression case `default` n'est jamais exécutée.

> **NOTE**
>
> Le compilateur n'émet pas d'avertissement pour signaler qu'une expression case `default` écrite ne peut pas s'exécuter. Cela est cohérent avec le comportement actuel de l'instruction `switch` où tous les cas possibles ont été répertoriés.

La troisième règle présente les cas où l'utilisation d'une expression case `var` peut être appropriée. Supposons que vous utilisiez des critères spéciaux dont l'entrée est une chaîne et que vous recherchiez des valeurs de commande connues. Vous pouvez écrire un code similaire à celui-ci :

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

L'instruction case `var` correspond à une valeur `null`, une chaîne vide ou une chaîne composée uniquement d'espaces blancs. On remarque que le code ci-dessus utilise l'opérateur `?.` pour empêcher la levée accidentelle d'une NullReferenceException. Le cas `default` gère toutes les autres valeurs de chaîne qui ne sont pas comprises par cet analyseur de commande.

Il s'agit d'un exemple où vous pouvez envisager d'utiliser une expression case `var` différente d'une expression `default`.

Conclusions

Les *constructions de critères spéciaux* permettent de gérer facilement le flux de contrôle entre différents types et variables qui ne sont pas liés par une hiérarchie d'héritage. Vous pouvez également contrôler la logique pour utiliser n'importe quelle condition testée sur la variable. Ces constructions permettent d'utiliser les modèles et les idiomes dont vous aurez davantage besoin à mesure que vous créez d'autres applications distribuées, où les données et les méthodes qui les manipulent sont séparées. Vous remarquerez que les structs de formes utilisés dans cet exemple ne contiennent pas de méthodes, mais seulement des propriétés en lecture seule. Les critères spéciaux peuvent être utilisés avec tous les types de données. Vous écrivez des expressions qui examinent l'objet et vous prenez des décisions de flux de contrôle en fonction de ces conditions.

Comparez le code de cet exemple avec la conception qui résulterait de la création d'une hiérarchie de classes pour une `Shape` abstraite et des formes dérivées spécifiques, chacune avec sa propre implémentation d'une méthode virtuelle pour calculer la surface. Les expressions de critères spéciaux vous seront souvent d'une grande utilité quand vous voudrez manipuler des données en gérant les questions de stockage de données séparément des questions de comportement.

Voir aussi

- Didacticiel : utiliser des critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données

Écrire du code C# sécurisé et efficace

22/04/2020 • 32 minutes to read • Edit Online

Les nouvelles fonctionnalités de C# vous permettent d'écrire du code sécurisé vérifiable avec de meilleures performances. Si vous appliquez soigneusement ces techniques, les scénarios nécessitant du code non sécurisé sont moins nombreux. Ces fonctionnalités facilitent l'utilisation des références à des types valeur comme arguments de méthode et retours de méthode. Quand elles sont utilisées de façon sécurisée, ces techniques réduisent la copie des types valeur. En utilisant des types valeur, vous pouvez réduire le nombre d'allocations et de passes du garbage collection.

Une grande partie de l'exemple de code de cet article utilise des fonctionnalités ajoutées dans C# 7.2. Pour utiliser ces fonctionnalités, vous devez configurer votre projet pour qu'il utilise C# 7.2 ou ultérieur. Pour plus d'informations sur la définition de la version linguistique, voir configurer la version linguistique.

Cet article met l'accent sur les techniques de gestion efficace des ressources. Un avantage de l'utilisation de types valeur est qu'ils permettent souvent d'éviter les allocations de tas. Mais l'inconvénient, c'est qu'ils sont copiés par valeur. Ce compromis rend plus difficile l'optimisation des algorithmes fonctionnaires sur de grandes quantités de données. Les nouvelles fonctionnalités du langage dans C# 7.2 fournissent des mécanismes qui permettent d'obtenir un code efficace sécurisé en utilisant des références à des types valeur. Utilisez ces fonctionnalités judicieusement pour minimiser à la fois les allocations et les opérations de copie. Cet article explore ces nouvelles fonctionnalités.

Cet article met l'accent sur les techniques de gestion des ressources suivantes :

- Déclarez `readonly struct` un pour exprimer qu'un type est **immuable**. Cela permet au compilateur d'enregistrer des copies défensives lors de l'utilisation de `in` paramètres.
- Si un type ne peut pas `struct` être `readonly` immuable, déclarez aux membres qu'ils indiquent que le membre ne modifie pas l'état.
- Utilisez `ref readonly` un rendement lorsque la `struct` valeur IntPtr.Size de retour est plus grande que la durée de vie de stockage est supérieure à la méthode de retour de la valeur.
- Quand la taille d'une `readonly struct` plus grande que IntPtr.Size, vous devez la passer comme paramètre `in` pour des raisons de performances.
- Ne passez `struct` jamais `in` un paramètre à moins `readonly` qu'il ne `readonly` soit déclaré avec le modificateur ou la méthode n'appelle que les membres de la struct. La violation de ces directives peut avoir une incidence négative sur les performances et pourrait conduire à un comportement obscur.
- Utilisez `ref struct` un, `readonly ref struct` ou Span<T> un ReadOnlySpan<T> tel ou de travailler avec la mémoire comme une séquence d'octets.

Ces techniques vous forcent à trouver un équilibre entre deux objectifs concurrents concernant les **références** et les **valeurs**. Les variables qui sont des types référence contiennent une référence à l'emplacement en mémoire. Les variables qui sont des types valeur contiennent directement leur valeur. Ces différences mettent en évidence les différences importantes pour la gestion des ressources mémoire. Les **types valeur** sont généralement copiés quand ils sont passés à une méthode ou retournés depuis une méthode. Ce comportement comprend la copie de la valeur de `this` lors de l'appel de membres d'un type valeur. Le coût de la copie est lié à la taille du type. Les **types référence** sont alloués sur le tas managé. Chaque nouvel objet nécessite une nouvelle allocation et doit par conséquent être récupéré par la suite. Ces deux opérations prennent du temps. La référence est copiée quand un type référence est passé comme argument à une méthode ou retourné depuis une méthode.

Cet article utilise l'exemple de concept suivant d'une structure de points 3D pour expliquer ces recommandations :

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Les différents exemples utilisent des implémentations différentes de ce concept.

Déclarer des structs en lecture seule pour les types valeur immuables

La déclaration d'une `struct` avec le modificateur `readonly` indique au compilateur que votre intention est de créer un type immuable. Le compilateur applique cette décision de conception avec les règles suivantes :

- Tous les champs membres doivent être `readonly`
- Toutes les propriétés doivent être en lecture seule, y compris les propriétés implémentées automatiquement.

Ces deux règles sont suffisantes pour garantir qu'aucun membre d'une `readonly struct` ne modifie l'état de ce struct. Le `struct` est immuable. La structure `Point3D` peut être définie comme struct immuable, comme le montre l'exemple suivant :

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Suivez cette recommandation quand l'intention de votre conception est de créer un type immuable. Toute amélioration des performances est un avantage supplémentaire. Le `readonly struct` exprime clairement l'intention de votre conception.

Déclarer les membres readonly quand une struct ne peut pas être immuable

Dans C 8.0 et plus tard, quand un type de struct est mutable, vous devez déclarer les membres qui ne causent pas la mutation d'être `readonly`. Considérez une application différente qui a besoin d'une structure de points 3D, mais doit soutenir la mutabilité. La version suivante de la structure `readonly` de points 3D ajoute le modificateur uniquement aux membres qui ne modifient pas la structure. Suivez cet exemple lorsque votre conception doit prendre en charge les modifications apportées à la structure par certains membres, mais vous voulez toujours les avantages de l'application de la lecture sur certains membres:

```
public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}
```

L'échantillon précédent montre de nombreux endroits `readonly` où vous pouvez appliquer le modificateur : méthodes, propriétés et accesseurs de propriété. Si vous utilisez des propriétés auto-mises `readonly` en `get` œuvre, le compilateur ajoute le modificateur à l'accesseur pour les propriétés de lecture-écriture. Le compilateur `readonly` ajoute le modificateur aux déclarations de `get` propriété auto-mises en œuvre pour les propriétés avec seulement un accesseur.

L'ajout du `readonly` modificateur aux membres qui ne mutent pas l'état fournit deux avantages connexes. Tout d'abord, le compilateur applique votre intention. Ce membre ne peut pas muter l'état de la structure. Deuxièmement, le compilateur ne créera pas de copies défensives des paramètres lors de `in` l'accès à un `readonly` membre. Le compilateur peut faire cette optimisation en `struct` toute sécurité car `readonly` il garantit que le n'est pas modifié par un membre.

Quand c'est possible, utilisez des instructions `ref readonly return` pour les grandes structures.

Vous pouvez retourner des valeurs par référence quand la valeur retournée n'est pas locale à la méthode effectuant le retour. Un retour par référence signifie que seule la référence est copiée, et non pas la structure. Dans l'exemple suivant, la propriété `Origin` ne peut pas utiliser un retour `ref`, car la valeur retournée est une variable locale :

```
public Point3D Origin => new Point3D(0,0,0);
```

Cependant, la définition de propriété suivante peut être retournée par référence, car la valeur retournée est un

membre statique :

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

Vous ne voulez pas que les appelants modifient la valeur l'origine : vous devez donc retourner la valeur par `ref readonly` :

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Retourner `ref readonly` vous permet d'éviter de copier des structures plus grandes et de préserver l'immuabilité de vos membres de données internes.

À l'endroit de l'appel, les appelants font le choix d'utiliser la propriété `Origin` comme `ref readonly` ou comme valeur :

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

La première assignation dans le code précédent effectue une copie de la constante `Origin` et assigne cette copie. La seconde assigne une référence. Notez que le modificateur `readonly` doit faire partie de la déclaration de la variable. La référence à laquelle il fait référence ne peut pas être modifiée. Toute tentative de modification provoque une erreur de compilation.

Le modificateur `readonly` est nécessaire sur la déclaration de `originReference`.

Le compilateur fait en sorte que l'appelant ne puisse pas modifier la référence. Toute tentative d'assignation directe de la valeur génère une erreur de compilation. Cependant, le compilateur ne peut pas savoir si une méthode membre modifie l'état du struct. Pour garantir que l'objet n'est pas modifié, le compilateur crée une copie et appelle les références de membre en utilisant cette copie. Les modifications sont apportées à cette copie défensive.

Appliquer le modificateur `in` à des paramètres `readonly struct` plus grands que `System.IntPtr.Size`

Le mot clé `in` vient en complément des mots clés `ref` et `out` existants pour passer des arguments par référence. Le mot clé `in` spécifie que l'argument doit être passé par référence, mais la méthode appelée ne modifie pas la valeur.

Vous disposez ainsi d'un vocabulaire complet pour exprimer votre intention de conception. Les types valeur sont copiés s'ils sont passés à une méthode appelée et que vous ne spécifiez pas l'un des modificateurs suivants dans

la signature de la méthode. Chacun de ces modificateurs spécifie qu'une variable est passée par référence, ce qui évite la copie. Chaque modificateur exprime une intention différente :

- `out` : Cette méthode définit la valeur de l'argument utilisé comme paramètre.
- `ref` : Cette méthode peut définir la valeur de l'argument utilisé comme paramètre.
- `in` : Cette méthode ne modifie pas la valeur de l'argument utilisé comme ce paramètre.

Ajoutez le modificateur `in` pour passer un argument par référence et déclarez que votre intention de conception est de passer des arguments par référence afin d'éviter toute copie inutile. Vous n'avez pas l'intention de modifier l'objet utilisé comme argument.

Cette pratique améliore souvent les performances pour les types valeur en lecture seule qui sont plus grands que IntPtr.Size. Pour les types simples (les types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` et `enum`), les gains de performances potentiels sont minimes. En fait, les performances peuvent se dégrader quand le passage par référence est utilisé pour les types plus petits que IntPtr.Size.

Le code suivant montre un exemple de méthode qui calcule la distance entre deux points dans l'espace 3D.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Les arguments sont deux structures qui contiennent chacune trois valeurs de type double. Un double représentant 8 octets, chaque argument représente 24 octets. En spécifiant le modificateur `in`, vous passez une référence de 4 ou 8 octets à ces arguments, selon l'architecture de la machine. La différence de taille est minime, mais elle peut s'accentuer si votre application appelle cette méthode dans une boucle dense avec de nombreuses valeurs différentes.

Le modificateur `in` complète aussi `out` et `ref` d'autres façons. Vous ne pouvez pas créer des surcharges de méthode qui diffèrent seulement par la présence de `in`, `out` ou `ref`. Ces nouvelles règles étendent le même comportement qui a toujours été défini pour les paramètres `out` et `ref`. Comme les modificateurs `out` et `ref`, les types valeur ne sont pas convertis (boxed), car le modificateur `in` est appliqué.

Le modificateur `in` peut être appliqué à tout membre acceptant des paramètres : méthodes, délégués, expressions lambda, fonctions locales, indexeurs, opérateurs.

Une autre caractéristique des paramètres `in` est que vous pouvez utiliser des valeurs littérales ou des constantes pour l'argument d'un paramètre `in`. Par ailleurs, contrairement à un paramètre `ref` ou `out`, il est inutile d'appliquer le modificateur `in` au niveau du site d'appel. Le code suivant donne deux exemples d'appel de la méthode `CalculateDistance`. Le premier utilise deux variables locales passées par référence. La deuxième inclut une variable temporaire créée dans le cadre de l'appel de méthode.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

Le compilateur garantit la nature « en lecture seule » d'un argument `in` de plusieurs façons. Premièrement, la méthode appelée ne peut rien assigner directement à un paramètre `in`. Elle ne peut rien assigner directement à un champ d'un paramètre `in` quand cette valeur est un type `struct`. De plus, vous ne pouvez pas passer un paramètre `in` à une méthode en utilisant le modificateur `ref` ou `out`. Ces règles s'appliquent à tout champ

d'un paramètre `in`, dans la mesure où le champ est un type `struct` et le paramètre également un type `struct`. En fait, ces règles s'appliquent à plusieurs couches d'accès au membre, dans la mesure où les types situés à tous les niveaux d'accès au membre sont des `structs`. Le compilateur considère que les types `struct` passés en tant qu'arguments `in` et leurs membres `struct` sont des variables en lecture seule quand ils sont utilisés en tant qu'arguments pour d'autres méthodes.

L'utilisation de paramètres `in` peut éviter les coûts potentiels en termes de performances qui sont liés aux copies. Elle ne change pas la sémantique des appels de méthode. Ainsi, vous n'avez pas besoin de spécifier le modificateur `in` à l'endroit de l'appel. L'omission du modificateur `in` à l'endroit de l'appel indique au compilateur qu'il est autorisé à effectuer une copie de l'argument pour les raisons suivantes :

- Il existe une conversion implicite, mais pas une conversion d'identité du type de l'argument en type du paramètre.
- L'argument est une expression, mais ne dispose pas d'une variable de stockage connue.
- Il existe une surcharge qui diffère selon la présence ou l'absence de `in`. Dans ce cas, la surcharge par valeur convient mieux.

Ces règles s'avèrent utiles quand vous mettez à jour le code existant pour utiliser des arguments de référence en lecture seule. À l'intérieur de la méthode appelée, vous pouvez appeler toute méthode d'instance qui utilise des paramètres par valeur. Dans ces instances, une copie du paramètre `in` est créée. Étant donné que le compilateur peut créer une variable temporaire pour un paramètre `in`, vous pouvez aussi spécifier des valeurs par défaut pour n'importe quel paramètre `in`. Le code suivant spécifie l'origine (point 0,0) comme valeur par défaut du deuxième point :

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Pour forcer le compilateur à passer des arguments en lecture seule par référence, spécifiez le modificateur `in` sur les arguments à l'endroit de l'appel, comme indiqué dans le code suivant :

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

Ce comportement facilite l'adoption de paramètres `in` au fil du temps dans les codes bases volumineux où des gains de performance sont possibles. Vous commencez par ajouter le modificateur `in` aux signatures de méthode. Ensuite, vous pouvez ajouter le modificateur `in` aux endroits des appels et créer des types `readonly struct` pour permettre au compilateur d'éviter de créer des copies défensives des paramètres `in` à d'autres emplacements.

La désignation du paramètre `in` peut également être utilisée avec des types référence ou des valeurs numériques. Toutefois, les avantages dans les deux cas sont minimes, voire inexistants.

Éviter les structs `in` mutables comme argument

Les techniques décrites ci-dessus expliquent comment éviter les copies en retournant des références et en passant les valeurs par référence. Ces techniques conviennent le mieux quand les types d'arguments sont déclarés comme types `readonly struct`. Sinon, le compilateur doit créer des **copies défensives** dans de nombreuses situations

pour garantir la nature « en lecture seule » des arguments. Considérons l'exemple suivant, qui calcule la distance d'un point 3D depuis l'origine :

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

La structure `Point3D` n'est *pas* un struct en lecture seule. Il y a six appels différents d'accès aux propriétés dans le corps de cette méthode. En première approche, vous pouvez penser que ces accès sont sécurisés. Après tout, un accesseur `get` ne devrait pas modifier l'état de l'objet. Mais il n'existe aucune règle du langage qui le garantisse. Il s'agit seulement d'une convention courante. N'importe quel type peut implémenter un accesseur `get` qui a modifié l'état interne. Sans une certaine garantie linguistique, le compilateur doit créer une copie `readonly` temporaire de l'argument avant d'appeler tout membre non marqué par le modificateur. Le stockage temporaire est créé sur la pile, les valeurs de l'argument sont copiées dans le stockage temporaire et la valeur est copiée sur la pile pour chaque accès au membre en tant qu'argument `this`. Dans de nombreuses situations, ces copies nuisent à des performances suffisantes pour que le passage par valeur `readonly struct` soit plus rapide que le `readonly` passage par la référence lorsque le type d'argument n'est pas un et la méthode appelle les membres qui ne sont pas marqués . Si vous marquez toutes les méthodes qui ne `readonly` modifient pas l'état de struct comme , le compilateur peut déterminer en toute sécurité que l'état de struct n'est pas modifié, et une copie défensive n'est pas nécessaire.

Au lieu de cela, si le calcul `ReadonlyPoint3D` de la distance utilise la struct immuable, les objets temporaires ne sont pas nécessaires:

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Le compilateur génère un code plus efficace `readonly struct` lorsque `this` vous appelez les membres d'un `in` : La référence, au lieu d'une copie du récepteur, est toujours un paramètre transmis en référence à la méthode du membre. Cette optimisation évite la copie quand vous utilisez un `readonly struct` comme argument `in`.

Vous ne devriez pas passer un type `in` de valeur nulle comme argument. Le Nullable<T> type n'est pas déclaré comme une struction de lecture seulement. Cela signifie que le compilateur doit générer des copies défensives pour tout argument de type valeur nullable passé à une méthode à l'aide du modificateur `in` sur la déclaration de paramètre.

Vous pouvez voir un programme d'exemple qui démontre les différences de performances à l'aide de BenchmarkDotNet dans notre référentiel d'échantillons sur GitHub. Il compare le passage d'un struct mutable par valeur et par référence au passage d'un struct immuable par valeur et par référence. L'utilisation du struct immuable et le passage par référence est le plus rapide.

Utilisez des types `ref struct` pour travailler avec des blocs ou de la mémoire sur un frame de pile

Une autre fonctionnalité associée du langage est la possibilité de déclarer un type valeur qui doit être contraint à un seul frame de pile. Cette restriction permet au compilateur d'effectuer plusieurs optimisations. Cette fonctionnalité vise principalement Span<T> et ses structures associées. Vous obtiendrez de meilleures performances de ces améliorations en utilisant les API .NET nouvelles et mises à jour, qui facilitent l'utilisation du type Span<T>.

Vous pouvez avoir des exigences `stackalloc` similaires en travaillant avec la mémoire créée en utilisant ou lors de l'utilisation de la mémoire des API interop. Pour répondre à ces besoins, vous pouvez définir vos propres types `ref struct`.

Type `readonly ref struct`

La déclaration d'un struct comme `readonly ref` combine les avantages et les restrictions des déclarations `ref struct` et `readonly struct`. La mémoire utilisée par l'étendue en lecture seule est limitée à un seul frame de pile, et cette mémoire ne peut pas être modifiée.

Conclusions

L'utilisation de types valeur réduit le nombre d'opérations d'allocation :

- Le stockage pour les types valeur est alloué sur la pile les variables locales et les arguments de méthode.
- Le stockage pour les types valeur qui sont membres d'autres objets est alloué dans le cadre de cet objet, et non pas comme allocation distincte.
- Le stockage pour les valeurs de retour de type valeur est alloué sur la pile.

Comparez cela avec les types référence dans ces mêmes situations :

- Le stockage pour les types référence est alloué sur le tas pour les variables locales et les arguments de méthode. La référence est stockée sur la pile.
- Le stockage pour les types référence qui sont membres d'autres objets sont alloués séparément sur le tas. L'objet conteneur stocke la référence.
- Le stockage des valeurs de retour de type référence est alloué sur le tas. La référence à ce stockage est stockée sur la pile.

La réduction des allocations entraîne des compromis. Vous copiez plus de mémoire quand la taille du `struct` est supérieure à la taille d'une référence. Une référence est généralement sur 64 ou 32 bits, et elle varie en fonction de l'UC de la machine cible.

Ces compromis ont généralement un impact minimal sur les performances. Cependant, pour les grands structs ou des collections plus grandes, l'impact sur les performances augmente. L'impact peut être important dans des boucles denses et des chemins d'exécution intensifs pour les programmes.

Ces améliorations apportées au langage C# sont conçues pour les algorithmes dont les performances sont critiques, dans lesquels la réduction des allocations mémoire est un facteur essentiel pour atteindre le niveau de performance nécessaire. Vous constaterez peut-être que vous n'utilisez pas souvent ces fonctionnalités dans votre code. Cependant, ces améliorations ont été adoptées dans .NET. Comme de plus en plus d'API adoptent ces fonctionnalités, vous verrez les performances de vos applications s'améliorer.

Voir aussi

- ref, mot clé
- Retours ref et variables locales ref

Arborescences de l'expression

02/11/2020 • 4 minutes to read • Edit Online

Si vous avez déjà utilisé LINQ, vous connaissez une bibliothèque enrichie où les types `Func` font partie de l'ensemble d'API. (Si vous n'êtes pas familiarisé avec LINQ, vous souhaiterez probablement lire le didacticiel LINQ et l'article sur les expressions lambda avant celui-ci.) Les *arborescences d'expressions* fournissent une interaction plus riche avec les arguments qui sont des fonctions.

Vous écrivez des arguments de fonction, généralement à l'aide d'expressions lambda, quand vous créez des requêtes LINQ. Dans une requête LINQ classique, ces arguments de fonction sont transformés en un délégué créé par le compilateur.

Quand vous souhaitez avoir une interaction plus complète, vous devez utiliser des *arborescences d'expressions*. Les arborescences d'expressions représentent le code sous forme de structure que vous pouvez examiner, modifier ou exécuter. Ces outils vous permettent de manipuler le code au moment de l'exécution. Vous pouvez écrire du code qui examine les algorithmes en cours d'exécution ou injecte de nouvelles fonctionnalités. Dans des scénarios avancés, vous pouvez modifier les algorithmes en cours d'exécution, et même traduire les expressions C# dans une autre forme en vue d'une exécution dans un autre environnement.

Vous avez probablement déjà écrit du code qui utilise des arborescences d'expressions. Les API LINQ d'Entity Framework acceptent des arborescences d'expressions comme arguments pour le modèle d'expression de requête LINQ. Cela permet à Entity Framework de traduire la requête que vous avez écrite en C# en code SQL qui s'exécute dans le moteur de base de données. Moq, framework de simulation populaire pour .NET, constitue un autre exemple.

Les sections suivantes de ce didacticiel expliquent en détail ce que sont les arborescences d'expressions, examinent les classes de framework qui prennent en charge les arborescences d'expressions et montrent comment utiliser des arborescences d'expressions. Vous découvrirez comment lire des arborescences d'expressions, créer des arborescences d'expressions, créer des arborescences d'expressions modifiées et exécuter le code représenté par des arborescences d'expressions. Une fois ce didacticiel terminé, vous serez prêt à utiliser ces structures pour créer des algorithmes adaptatifs enrichis.

1. Explication des arborescences d'expressions

   Comprenez la structure et les concepts derrière les *arborescences d'expressions*.

2. Types de frameworks prenant en charge les arborescences d'expressions

   Découvrez les structures et les classes qui définissent et manipulent des arborescences d'expressions.

3. Exécution des expressions

   Découvrez comment convertir une arborescence d'expressions représentée sous forme d'expression lambda en un délégué, et comment exécuter le délégué résultant.

4. Interprétation des expressions

   Découvrez comment parcourir et examiner des *arborescences d'expressions* pour comprendre quel est le code représenté par l'arborescence d'expressions.

5. Création des expressions

   Découvrez comment construire les nœuds d'une arborescence d'expressions et comment générer des arborescences d'expressions.

6. Traduction des expressions

   Découvrez comment générer une copie modifiée d'une arborescence d'expressions ou comment traduire une arborescence d'expressions dans un format différent.

7. Résumé

   Passez en revue les informations sur les arborescences d'expressions.

Explication des arborescences d'expressions

18/03/2020 • 9 minutes to read • Edit Online

Précédent – Vue d'ensemble

Les arborescences d'expressions sont des structures de données qui définissent du code. Elles sont basées sur les mêmes structures que celles utilisées par un compilateur pour analyser du code et générer la sortie compilée. Lors de la lecture de ce didacticiel, vous remarquerez qu'il existe certaines similitudes entre les arborescences d'expressions et les types utilisés dans les API Roslyn pour générer des Analyzers et CodeFixes. (Les analyseurs et les codeFixes sont des paquets NuGet qui effectuent une analyse statique sur le code et peuvent suggérer des correctifs potentiels pour un développeur.) Les concepts sont similaires, et le résultat final est une structure de données qui permet l'examen du code source d'une manière significative. Toutefois, les arborescences d'expressions sont basées sur un ensemble de classes et d'API totalement différentdes API Roslyn.

Examinons un exemple simple. Voici une ligne de code :

```
var sum = 1 + 2;
```

Si vous analysez ceci comme une arborescence d'expressions, l'arborescence contient plusieurs nœuds. Le nœud extérieur est une instruction de déclaration de variable avec attribution (`var sum = 1 + 2;`). Ce nœud extérieur contient plusieurs nœuds enfants : une déclaration de variable, un opérateur d'assignation et une expression qui représente la partie à droite du signe égal. Cette expression est sous-divisée en expressions qui représentent l'opération d'addition et les opérandes gauche et droit de l'addition.

Penchons-nous un peu plus sur les expressions qui composent la partie à droite du signe égal. L'expression est `1 + 2`. Il s'agit d'une expression binaire. Plus spécifiquement, il s'agit d'une expression d'addition binaire. Une expression d'addition binaire a deux enfants, représentant les nœuds gauche et droit de l'expression d'addition. Ici, les deux nœuds sont des expressions constantes : l'opérande gauche est la valeur `1`, et l'opérande droit est la valeur `2`.

Visuellement, l'instruction entière est une arborescence. Vous pouvez partir du nœud racine et accéder à chaque nœud dans l'arborescence pour voir le code qui compose l'instruction :

- Instruction de déclaration de variable avec attribution (`var sum = 1 + 2;`)
  - Déclaration de type de variable implicite (`var sum`)
    - Mot clé var implicite (`var`)
    - Déclaration de nom de variable (`sum`)
  - Opérateur d'assignation (`=`)
  - Expression d'addition binaire (`1 + 2`)
    - Opérande gauche (`1`)
    - Opérateur d'addition (`+`)
    - Opérande droit (`2`)

Cela peut sembler compliqué, mais c'est très puissant. Suivant le même processus, vous pouvez décomposer des expressions beaucoup plus complexes. Examinons cette expression :

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

L'expression ci-dessus est aussi une déclaration de variable avec une assignation. La partie droite de l'assignation est une arborescence beaucoup plus complexe. Je ne vais pas décomposer cette expression, mais réfléchissez à ce que pourraient être les différents nœuds. Il y a des appels de méthode utilisant l'objet actif comme destinataire, un qui a un récepteur `this` explicite, un autre qui n'en a pas. Il y a des appels de méthode utilisant d'autres objets récepteurs, et des arguments constants de différents types. Pour finir, il y a un opérateur d'addition binaire. En fonction du type de retour de `SecretSauceFunction()` ou `MoreSecretSauce()`, cet opérateur d'addition binaire peut être un appel de méthode à un opérateur d'addition substitué, résolu en un appel de méthode statique à l'opérateur d'addition binaire défini pour une classe.

Malgré cette complexité apparente, l'expression ci-dessus crée une arborescence qui peut être parcourue aussi facilement que le premier exemple. Vous pouvez traverser les nœuds enfants successifs pour rechercher des nœuds terminaux dans l'expression. Les nœuds parents ont des références à leurs enfants, et chaque nœud a une propriété qui décrit le genre de nœud dont il s'agit.

La structure d'une arborescence d'expressions est très cohérente. Une fois que vous maîtrisez les principes de base, vous pouvez comprendre le code le plus complexe quand il est représenté sous forme d'arborescence d'expressions. L'élégance dans la structure de données explique comment le compilateur C# peut analyser les programmes C# les plus complexes et créer une sortie appropriée à partir de ce code source complexe.

Une fois que vous vous serez familiarisé avec la structure des arborescences d'expressions, vous constaterez que les connaissances acquises vous permettront de travailler rapidement avec de nombreux scénarios avancés. Les arborescences d'expressions offrent une puissance incroyable.

Outre la conversion d'algorithmes à exécuter dans d'autres environnements, vous pouvez utiliser des arborescences d'expressions pour simplifier l'écriture d'algorithmes qui inspectent du code avant de l'exécuter. Vous pouvez écrire une méthode dont les arguments sont des expressions, puis examiner ces expressions avant d'exécuter le code. L'arborescence d'expressions est une représentation complète du code : vous pouvez voir les valeurs de n'importe quelle sous-expression. Vous pouvez voir les noms des méthodes et des propriétés. Vous pouvez voir la valeur de n'importe quelle expression constante. Vous pouvez également convertir une arborescence d'expressions en délégué exécutable et exécuter le code.

Les API d'arborescences d'expressions vous permettent de créer des arborescences qui représentent presque n'importe quelle construction de code valide. Toutefois, pour simplifier les choses au maximum, certains idiomes C# ne peuvent pas être créés dans une arborescence d'expressions. C'est le cas des expressions asynchrones (utilisant les mots clés `async` et `await`). Si vous avez besoin d'algorithmes asynchrones, vous devez manipuler directement les objets `Task` plutôt que de vous fier à la prise en charge du compilateur. C'est aussi le cas pour la création des boucles. En général, vous les créez à l'aide de boucles `for`, `foreach`, `while` ou `do`. Comme vous le verrez plus loin dans cette série, les API d'arborescences d'expressions prennent en charge une expression de boucle unique, avec des expressions `break` et `continue` qui contrôlent la répétition de la boucle.

La seule chose que vous ne pouvez pas faire est de modifier une arborescence d'expressions. Les arborescences d'expressions sont des structures de données immuables. Si vous souhaitez muter (changer) une expression de l'arborescence, vous devez créer une nouvelle arborescence qui est une copie de l'original, mais avec les modifications souhaitées.

Suivant – Types de frameworks prenant en charge les arborescences d'expressions

Types de frameworks prenant en charge les arborescences d'expressions

02/11/2020 • 5 minutes to read • Edit Online

Précédent – Explication des arborescences d'expressions

Il existe une longue liste de classes dans le framework .NET Core qui fonctionnent avec des arborescences d'expressions. Vous pouvez voir la liste complète dans System.Linq.Expressions. Au lieu de parcourir la liste complète, essayons de comprendre comment les classes du framework ont été conçues.

Dans la conception du langage, une expression est un corps de code qui évalue et retourne une valeur. Les expressions peuvent être très simples : l'expression constante `1` retourne la valeur constante 1. Elles peuvent être plus compliquées : l'expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` retourne une racine pour une équation quadratique (dans le cas où l'équation a une solution).

Tout commence par System.Linq.Expression

L'une des complexités liées à l'utilisation des arborescences d'expressions est que de nombreuses sortes d'expressions sont valides à de nombreux endroits dans les programmes. Prenez par exemple une expression d'assignation. La partie droite d'une assignation peut être une valeur constante, une variable, une expression d'appel de méthode ou autre. Cette flexibilité linguistique signifie que vous pouvez rencontrer de nombreux types d'expressions différents à n'importe quel endroit dans les nœuds d'une arborescence quand vous parcourez une arborescence d'expressions. Ainsi, quand vous pouvez travailler avec le type d'expression de base, c'est le moyen le plus simple de procéder. Cependant, vous devez parfois en savoir plus. La classe Expression de base contient une propriété `NodeType` à cet effet. Elle retourne un `ExpressionType` qui est une énumération des types d'expressions possibles. Une fois que vous connaissez le type du nœud, vous pouvez le caster en ce type et effectuer des actions spécifiques. Vous pouvez rechercher certains types de nœuds, puis utiliser les propriétés spécifiques de ce genre d'expression.

Par exemple, ce code imprime le nom d'une variable pour une expression d'accès à la variable. J'ai suivi la pratique qui consiste à vérifier le type de nœud, à caster en une expression d'accès à la variable, puis à vérifier les propriétés du type d'expression spécifique :

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Création d'arborescences d'expressions

La classe `System.Linq.Expression` contient également de nombreuses méthodes statiques pour créer des expressions. Ces méthodes créent un nœud d'expression en utilisant les arguments fournis pour ses enfants. De cette façon, vous générez une expression à partir de ses nœuds terminaux. Par exemple, ce code génère une expression Add :

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

Vous pouvez constater d'après cet exemple simple que de nombreux types sont impliqués dans la création et l'utilisation des arborescences d'expressions. Cette complexité est nécessaire pour fournir les fonctionnalités du vocabulaire enrichi fourni par le langage C#.

Navigation dans les API

Il existe des types de nœuds Expression qui mappent à presque tous les éléments de syntaxe du langage C#. Chaque type a des méthodes spécifiques pour ce type d'élément de langage. Cela fait beaucoup de choses à mémoriser en même temps. Plutôt que d'essayer de mémoriser tout, voici les techniques que je vous conseille d'appliquer quand vous travaillez avec des arborescences d'expressions :

1. Examinez les membres de l'énumération `ExpressionType` pour déterminer les éventuels nœuds à examiner. Cela aide vraiment quand vous souhaitez parcourir et comprendre une arborescence d'expressions.
2. Examinez les membres statiques de la classe `Expression` pour générer une expression. Ces méthodes peuvent générer tout type d'expression à partir d'un ensemble de ses nœuds enfants.
3. Examinez la classe `ExpressionVisitor` pour générer une arborescence d'expressions modifiée.

Vous trouverez davantage de choses à mesure que vous examinez chacune de ces trois zones. Vous trouverez invariablement ce dont vous avez besoin si vous commencez avec l'une de ces trois étapes.

Suivant – Exécution d'arborescences d'expressions

Exécution d'arborescences d'expressions

18/03/2020 • 11 minutes to read • Edit Online

Précédent -- Types de frameworks prenant en charge les arborescences d'expressions

Une *arborescence d'expressions* est une structure de données qui représente du code. Il ne s'agit pas de code compilé et exécutable. Si vous souhaitez exécuter le code .NET représenté par une arborescence d'expressions, vous devez le convertir en instructions de langage intermédiaire exécutables.

Des expressions lambda aux fonctions

Vous pouvez convertir n'importe quelle LambdaExpression ou n'importe quel type dérivé de LambdaExpression en langage intermédiaire exécutable. Les autres types d'expressions ne peuvent pas être convertis directement en code. Cette restriction a peu d'effet dans la pratique. Les expressions lambda sont les seuls types d'expressions que vous pourriez souhaiter exécuter par l'intermédiaire d'une conversion en langage intermédiaire exécutable. (Pensez à ce que cela signifierait d'exécuter directement une `ConstantExpression`. Cela signifierait-il quelque chose d'utile?) Tout arbre d'expression qui est `LambdaExpression` `LambdaExpression` un , ou un type dérivé peut être converti en IL. Le type d'expression `Expression<TDelegate>` est le seul exemple concret dans les bibliothèques .NET Core. Il sert à représenter une expression mappée à un type délégué quelconque. Ce type étant mappé à un type délégué, .NET peut examiner l'expression et générer le langage intermédiaire pour un délégué approprié qui correspond à la signature de l'expression lambda.

Dans la plupart des cas, cela crée un mappage simple entre une expression et son délégué correspondant. Par exemple, une arborescence d'expressions représentée par `Expression<Func<int>>` serait convertie en un délégué du type `Func<int>`. Pour une expression lambda avec tout type de retour et liste d'arguments, il existe un type délégué qui est le type cible pour le code exécutable représenté par cette expression lambda.

Le type `LambdaExpression` contient des membres `Compile` et `CompileToMethod` que vous utiliseriez pour convertir une arborescence d'expressions en code exécutable. La méthode `Compile` crée un délégué. La méthode `CompileToMethod` met à jour un objet `MethodBuilder` avec le langage intermédiaire qui représente la sortie compilée de l'arborescence d'expressions. Notez que `CompileToMethod` n'est disponible que sur le framework de bureau complet, et non dans .NET Core.

Si vous le souhaitez, vous pouvez également fournir un `DebugInfoGenerator` qui recevra les informations de débogage de symbole pour l'objet délégué généré. Cela vous permet de convertir l'arborescence d'expressions en objet délégué et de disposer d'informations de débogage complètes sur le délégué généré.

Vous pouvez convertir une expression en délégué à l'aide du code suivant :

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Notez que le type délégué est basé sur le type d'expression. Si vous souhaitez utiliser l'objet délégué d'une manière fortement typée, vous devez connaître le type de retour et la liste d'arguments. La méthode `LambdaExpression.Compile()` retourne le type `Delegate`. Vous devrez effectuer un cast vers le type délégué approprié pour que les outils de compilation puissent vérifier la liste d'arguments de type de retour.

Exécution et durées de vie

Vous exécutez le code en appelant le délégué créé quand vous avez appelé `LambdaExpression.Compile()`. Vous pouvez l'observer ci-dessus, où `add.Compile()` retourne un délégué. L'appel de ce délégué en appelant `func()` exécute le code.

Ce délégué représente le code dans l'arborescence d'expressions. Vous pouvez conserver le handle de ce délégué et l'appeler ultérieurement. Vous n'avez pas besoin de compiler l'arborescence d'expressions chaque fois que vous souhaitez exécuter le code qu'elle représente. (N'oubliez pas que les arborescences d'expressions sont immuables ; compiler la même arborescence d'expressions ultérieurement crée un délégué qui exécute le même code.)

Je vous déconseille d'essayer de créer des mécanismes de mise en cache plus complexes pour améliorer les performances en évitant les appels de compilation inutiles. Comparer deux arborescences d'expressions arbitraires pour déterminer si elles représentent le même algorithme prend également beaucoup de temps. Vous vous rendrez sans doute compte que le temps de calcul économisé en évitant tout appel supplémentaire à `LambdaExpression.Compile()` sera largement contrebalancé par le temps passé à exécuter le code qui détermine si deux arborescences d'expressions génèrent le même code exécutable.

Mises en garde

Compiler une expression lambda en un délégué et appeler ce délégué sont l'une des opérations les plus simples que vous puissiez effectuer avec une arborescence d'expressions. Toutefois, même avec cette simple opération, il existe des pièges dont vous devez être conscient.

Les expressions lambda créent des fermetures sur toutes les variables locales qui sont référencées dans l'expression. Vous devez vérifier que toutes les variables qui font partie du délégué sont utilisables à l'emplacement où vous appelez `Compile` et quand vous exécutez le délégué résultant.

En règle générale, le compilateur s'assure que ce soit le cas. Toutefois, si votre expression accède à une variable qui implémente `IDisposable`, il est possible que votre code supprime l'objet alors qu'il est toujours détenu par l'arborescence d'expressions.

Par exemple, ce code fonctionne bien, car `int` n'implémente pas `IDisposable` :

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

Le délégué a capturé une référence à la variable locale `constant`. Cette variable est accessible à tout moment ultérieurement, quand la fonction retournée par `CreateBoundFunc` s'exécute.

Maintenant, examinez cette classe qui implémente `IDisposable` :

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

Si vous l'utilisez dans une expression comme illustré ci-dessous, vous obtenez une `ObjectDisposedException` quand vous exécutez le code référencé par la propriété `Resource.Argument` :

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

Le délégué retourné par cette méthode a fermé l'objet `constant`, qui a été supprimé. (Il a été supprimé car il a été déclaré dans une instruction `using`.)

Désormais, quand vous exécuterez le délégué retourné par cette méthode, une `ObjectDisposedException` sera levée au moment de l'exécution.

Cela semble étrange d'avoir une erreur d'exécution qui représente une construction de compilation, mais cela fait partie des éventualités quand nous travaillons avec des arborescences d'expressions.

Comme il existe de nombreuses permutations de ce problème, il est difficile de proposer des conseils généraux pour l'éviter. Faites attention si vous accédez à des variables locales quand vous définissez des expressions et faites attention si vous accédez à l'état de l'objet actif (représenté par `this`) quand vous créez une arborescence d'expressions qui peut être retournée par une API publique.

Le code dans votre expression peut référencer des méthodes ou des propriétés dans d'autres assemblys. Ces assemblys doivent être accessibles quand l'expression est définie, quand elle est compilée et quand le délégué résultant est appelé. S'il n'est pas présent, vous recevrez une `ReferencedAssemblyNotFoundException`.

Résumé

Les arborescences d'expressions qui représentent des expressions lambda peuvent être compilées pour créer un délégué exécutable. Cela fournit un mécanisme pour exécuter le code représenté par une arborescence d'expressions.

L'arborescence d'expressions représente le code qui s'exécuterait pour toute construction donnée que vous créez. Tant que l'environnement dans lequel vous compilez et exécutez le code correspond à celui dans lequel vous créez une expression, tout fonctionne comme prévu. Si ce n'est pas le cas, les erreurs sont très prévisibles et seront

interceptées lors de vos premiers tests de n'importe quel code utilisant les arborescences d'expressions.

Suivant -- Interprétation d'expressions

Interprétation des expressions

02/11/2020 • 20 minutes to read • Edit Online

Précédent -- Exécution d'expressions

Nous allons maintenant écrire du code pour analyser la structure d'une *arborescence d'expressions*. Chaque nœud d'une arborescence d'expressions est un objet d'une classe dérivée de `Expression`.

Grâce à cette conception, visiter tous les nœuds d'une arborescence d'expressions est une opération récursive relativement simple. La stratégie générale consiste à débuter au niveau du nœud racine et à déterminer de quel genre de nœud il s'agit.

Si le type de nœud a des enfants, visitez-les de manière récursive. Sur chaque nœud enfant, répétez le processus utilisé au niveau du nœud racine : déterminez le type et, s'il a des enfants, visitez chacun d'eux.

Examen d'une expression sans enfants

Commençons par visiter chaque nœud dans une arborescence d'expressions simple. Voici le code qui crée une expression constante, puis examine ses propriétés :

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

La sortie suivante est générée :

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Maintenant, nous allons écrire le code qui examinerait cette expression et écrirait certaines propriétés importantes à son sujet. Voici ce code :

Examen d'une expression d'addition simple

Commençons par l'exemple d'addition issu de l'introduction à cette section.

```
Expression<Func<int>> sum = () => 1 + 2;
```

> Je n'utilise pas `var` pour déclarer cette arborescence d'expressions. C'est impossible, car la partie droite de l'assignation est implicitement typée.

Le nœud racine est `LambdaExpression`. Pour obtenir le code intéressant sur le côté droit de l' `=>` opérateur, vous devez Rechercher l'un des enfants de `LambdaExpression` . Nous allons le faire avec toutes les expressions dans cette section. Le nœud parent nous aide à trouver le type de retour de `LambdaExpression`.

Pour examiner chaque nœud de cette expression, nous devons visiter plusieurs nœuds de manière récursive. Voici une première implémentation simple :

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type: {argumentExpression.Type.ToString()}, Name:
{argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");
```

Cet exemple génère la sortie suivante :

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

Vous remarquerez que l'exemple de code ci-dessus contient un grand nombre de répétitions. Nous allons le nettoyer et générer un visiteur de nœud d'expression plus général. Pour cela, nous allons devoir écrire un algorithme récursif. Tout nœud peut être d'un type susceptible d'avoir des enfants. Tout nœud ayant des enfants nous oblige à visiter ces enfants et à déterminer ce qu'est ce nœud. Voici la version nettoyée qui utilise la récursivité pour visiter les opérations d'addition :

```
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
```

```
                return new ParameterVisitor((ParameterExpression)node);
            case ExpressionType.Add:
                return new BinaryVisitor((BinaryExpression)node);
            default:
                Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
                return default(Visitor);
        }
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name == null) ? "<null>" :
node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }
```

```
    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}
```

Cet algorithme est la base d'un algorithme qui peut visiter toute `LambdaExpression` arbitraire. Il existe de nombreux trous, à savoir que le code que j'ai créé recherche uniquement un très petit échantillon des jeux possibles de nœuds d'arborescence d'expressions qu'il peut rencontrer. mais ce qu'il génère peut quand même nous en apprendre beaucoup. (Le cas par défaut dans la méthode `Visitor.CreateFromExpression` affiche un message dans la console d'erreurs quand un nouveau type de nœud est détecté. De cette façon, vous savez que vous devez ajouter un nouveau type d'expression.)

Quand nous exécutons ce visiteur sur l'expression d'addition ci-dessus, nous obtenons la sortie suivante :

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Maintenant que nous avons créé une implémentation de visiteur plus générale, nous pouvons visiter et traiter de nombreux autres types d'expressions.

Examen d'une expression d'addition avec de nombreux niveaux

Essayons un exemple plus complexe, tout en continuant de limiter les types de nœuds à l'addition uniquement :

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Avant d'exécuter cet exemple sur l'algorithme visiteur, essayez de réfléchir à ce que pourrait être le résultat.

N'oubliez pas que l'opérateur `+` est un *opérateur binaire* : il doit avoir deux enfants, représentant les opérandes gauche et droit. Il existe plusieurs manières correctes de construire une arborescence :

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Vous pouvez voir la séparation en deux réponses possibles, pour mettre en évidence la plus prometteuse. La première représente les expressions *associatives à droite*. La deuxième représente les expressions *associatives à gauche*. L'avantage de ces deux formats est qu'ils peuvent s'adapter à un nombre arbitraire d'expressions d'addition.

Si vous exécutez cette expression par l'intermédiaire du visiteur, vous verrez cette sortie, en vérifiant que l'expression d'addition simple est *associative à gauche*.

Pour exécuter cet exemple et voir l'arborescence d'expressions complète, j'ai dû apporter une modification à l'arborescence d'expressions source. Quand l'arborescence d'expressions contient uniquement des constantes, l'arborescence résultante contient simplement la valeur constante `10`. Le compilateur effectue toute l'addition et réduit l'expression à sa forme la plus simple. Le simple ajout d'une variable dans l'expression suffit pour voir l'arborescence d'origine :

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Créez un visiteur pour cette somme et exécutez-le. Vous obtiendrez cette sortie :

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

Vous pouvez également exécuter l'un des autres exemples dans le code de visiteur et voir quelle arborescence il représente. Voici un exemple de l'expression `sum3` ci-dessus (avec un paramètre supplémentaire pour empêcher le compilateur de calculer la constante) :

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Voici la sortie du visiteur :

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Notez que les parenthèses ne font pas partie de la sortie. Il n'y a aucun nœud dans l'arborescence d'expressions qui représente les parenthèses dans l'expression d'entrée. La structure de l'arborescence d'expressions contient toutes les informations nécessaires pour communiquer le niveau de priorité.

Extension de cet exemple

L'exemple traite uniquement les arborescences d'expressions les plus rudimentaires. Le code que nous avons vu dans cette section gère seulement des entiers constants et l'opérateur binaire `+`. En guise de dernier exemple, nous allons mettre à jour le visiteur pour gérer une expression plus complexe. Prenons le code ci-dessous :

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Il représente une implémentation possible de la fonction mathématique *factorial*. La façon dont ce code est écrit met en évidence deux limitations concernant la méthode de création d'arborescences d'expressions qui consiste à assigner des expressions lambda à Expression. Tout d'abord, les lambda-instructions ne sont pas autorisées. Cela signifie que je ne peux pas utiliser de boucles, de blocs, d'instructions if / else et d'autres structures de contrôle courantes en C#. Je suis limité à l'utilisation d'expressions. Ensuite, je ne peux pas appeler la même expression de manière récursive. Je le pourrais s'il s'agissait déjà d'un délégué, mais je ne peux pas l'appeler sous sa forme d'arborescence d'expressions. Dans la section relative à la création d'arborescences d'expressions, vous apprendrez les techniques permettant de surmonter ces limitations.

Dans cette expression, vous rencontrerez des nœuds de tous ces types :

1. Égal (expression binaire)
2. Multiplier (expression binaire)
3. Conditionnel (l'expression ? :)
4. Expression d'appel de méthode (appel de `Range()` et `Aggregate()`)

L'une des manières de modifier l'algorithme visiteur consiste à continuer à l'exécuter, et à écrire le type de nœud chaque fois que vous atteignez votre clause `default`. Après quelques itérations, vous aurez vu chacun des nœuds potentiels. Vous aurez alors tout ce qu'il vous faut. Le résultat ressemblera à ceci :

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

ConditionalVisitor et MethodCallVisitor traitent ces deux nœuds :

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach(var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

Et la sortie de l’arborescence d’expressions sera :

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef: False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef: False
                        The expression body is:
                                This binary expression is a Multiply expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product, ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor, ByRef: False
```

Extension de la bibliothèque d'exemples

Les exemples de cette section illustrent les techniques de base pour visiter et examiner des nœuds dans une arborescence d'expressions. Si j'ai ignoré de nombreuses actions dont vous pourriez avoir besoin, c'est pour mieux me concentrer sur les tâches fondamentales de visite et d'accès aux nœuds dans une arborescence d'expressions.

Tout d'abord, les visiteurs gèrent uniquement les constantes qui sont des entiers. Les valeurs constantes peuvent être n'importe quel autre type numérique, et le langage C# prend en charge les conversions et les promotions entre ces types. Une version plus robuste de ce code mettrait en miroir toutes ces capacités.

Même le dernier exemple reconnaît un sous-ensemble des types de nœuds possibles. Vous pouvez toujours lui fournir de nombreuses expressions qui entraîneront son échec. Une implémentation complète est incluse dans

.NET Standard sous le nom ExpressionVisitor et peut gérer tous les types de nœuds possibles.

Pour finir, la bibliothèque que j'ai utilisée dans cet article a été créée à des fins de démonstration et de formation. Elle n'est pas optimisée. Je l'ai écrit pour rendre les structures utilisées de façon claire, et pour mettre en évidence les techniques utilisées pour visiter les nœuds et analyser ce qu'il y a. Une implémentation de production accorderait davantage d'attention aux performances.

Même avec ces limitations, vous devriez être sur la bonne voie pour écrire des algorithmes qui lisent et comprennent les arborescences d'expressions.

Suivant -- Génération d'expressions

Génération d'arborescences d'expressions

02/11/2020 • 9 minutes to read • Edit Online

Précédent -- Interprétation d'expressions

Toutes les arborescences d'expressions que vous avez vues jusqu'ici ont été créées par le compilateur C#. Tout ce que vous aviez à faire était de créer une expression lambda qui était assignée à une variable de type `Expression<Func<T>>` ou d'un type similaire. Ce n'est pas la seule façon de créer une arborescence d'expressions. Dans de nombreux scénarios, vous constaterez peut-être que vous devez générer une expression en mémoire au moment de l'exécution.

La génération d'arborescences d'expressions est compliquée par le fait qu'elles sont immuables. Cela signifie que vous devez générer l'arborescence à partir des feuilles jusqu'à la racine. Les API que vous utiliserez pour générer des arborescences d'expressions reflètent cet état de fait : les méthodes que vous utiliserez pour générer un nœud prennent tous ses enfants en tant qu'arguments. Examinons quelques exemples illustrant les techniques à appliquer.

Création de nœuds

Commençons par quelque chose de relativement simple. Nous allons utiliser l'expression d'addition avec laquelle j'ai travaillé tout au long de ces sections :

```
Expression<Func<int>> sum = () => 1 + 2;
```

Pour construire cette arborescence d'expressions, nous devons construire les nœuds terminaux. Les nœuds terminaux étant des constantes, nous pouvons utiliser la méthode `Expression.Constant` pour créer les nœuds :

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Ensuite, nous allons générer l'expression d'addition :

```
var addition = Expression.Add(one, two);
```

Après cela, nous pouvons créer l'expression lambda :

```
var lambda = Expression.Lambda(addition);
```

Cette expression lambda est très simple, car elle ne contient aucun argument. Plus loin dans cette section, nous verrons comment mapper des arguments à des paramètres, et comment générer des expressions plus complexes.

Pour les expressions qui sont aussi simples que celle-ci, nous pouvons combiner tous les appels en une seule instruction :

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Génération d'une arborescence

Voilà les principes fondamentaux de génération d'une arborescence d'expressions en mémoire. Les arborescences plus complexes signifient généralement qu'il y a davantage de types de nœuds et davantage de nœuds dans l'arborescence. Examinons un autre exemple et voyons deux autres types de nœuds que vous générerez souvent lors de la création d'arborescences d'expressions : les nœuds d'argument et les nœuds d'appel de méthode.

Commençons par générer une arborescence d'expressions pour créer cette expression :

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Nous allons commencer par créer des expressions de paramètre pour `x` et `y` :

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

La création des expressions de multiplication et d'addition suit le modèle que nous avons déjà vu :

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Ensuite, nous devons créer une expression d'appel de méthode pour l'appel à `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });
var distance = Expression.Call(sqrtMethod, sum);
```

Pour finir, nous plaçons l'appel de méthode dans une expression lambda et nous veillons à définir les arguments de l'expression lambda :

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

Dans cet exemple plus complexe, vous pouvez observer quelques techniques supplémentaires dont vous aurez souvent besoin pour créer des arborescences d'expressions.

Tout d'abord, vous devez créer les objets qui représentent des variables locales ou des paramètres avant de les utiliser. Une fois que vous avez créé ces objets, vous pouvez les utiliser dans votre arborescence d'expressions partout où vous en avez besoin.

Ensuite, vous devez utiliser un sous-ensemble des API de réflexion pour créer un objet `MethodInfo` afin de pouvoir créer une arborescence d'expressions pour accéder à cette méthode. Vous devez vous limiter au sous-ensemble des API de réflexion disponibles sur la plateforme .NET Core. Là encore, ces techniques s'étendront à

d'autres arborescences d'expressions.

Code de génération en profondeur

Vous n'êtes pas limité dans ce que vous pouvez générer à l'aide de ces API. Toutefois, plus l'arborescence d'expressions que vous souhaitez générer est compliquée, plus le code est difficile à gérer et à lire.

Nous allons générer une arborescence d'expressions qui est l'équivalent de ce code :

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Vous remarquerez ci-dessus que je n'ai pas généré l'arborescence d'expressions, mais simplement le délégué. À l'aide de la classe `Expression`, vous ne pouvez pas créer de lambda-instructions. Voici le code nécessaire pour générer la même fonctionnalité. Il est compliqué par le fait qu'il n'existe pas d'API pour générer une boucle `while`. Au lieu de cela, vous devez générer une boucle qui contient un test conditionnel, et une cible d'étiquette pour quitter la boucle.

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

Le code de génération de l'arborescence d'expressions pour la fonction factorielle est un peu plus long, plus compliqué et truffé d'étiquettes, d'instructions break et d'autres éléments que nous souhaitons éviter lors de nos tâches quotidiennes de codage.

Dans cette section, j'ai également mis à jour le code de visiteur pour visiter tous les nœuds de cette arborescence

d'expressions et écrire des informations sur les nœuds créés dans cet exemple. Vous pouvez afficher ou télécharger l'exemple de code à partir du dépôt GitHub dotnet/docs. Essayez par vous-même en créant et en exécutant les exemples. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Examen des API

Les API d'arborescence d'expressions comptent parmi les plus difficiles à parcourir dans .NET Core, mais ce n'est pas grave. Leur objectif est plutôt complexe : écrire du code qui génère du code au moment de l'exécution. Elles sont nécessairement compliquées afin de fournir un équilibre entre la prise en charge de toutes les structures de contrôle disponibles en langage C# et la taille de la surface d'exposition des API (qui doit être la plus petite possible). Cet équilibre signifie que de nombreuses structures de contrôle sont représentées non pas par leurs constructions C#, mais par des constructions qui représentent la logique sous-jacente générée par le compilateur à partir de ces constructions de niveau supérieur.

Il existe aussi actuellement des expressions C# qui ne peuvent pas être créées directement à l'aide de méthodes de classe `Expression`. En général, il s'agit des opérateurs et expressions les plus récents ajoutés dans C# 5 et C# 6. (Par exemple, les expressions `async` ne peuvent pas être générées et le nouvel opérateur `?.` ne peut pas être créé directement.)

Suivant -- Traduction d'expressions

Traduire les arborescences d'expressions

02/11/2020 • 10 minutes to read • Edit Online

Précédent -- Génération d'expressions

Dans cette dernière section, vous allez découvrir comment visiter chaque nœud dans une arborescence d'expressions lors de la génération d'une copie modifiée de cette arborescence d'expressions. Voici les techniques que vous utiliserez dans deux scénarios importants. La première consiste à comprendre les algorithmes exprimés par une arborescence d'expressions pour qu'elle puisse être traduite dans un autre environnement. La deuxième s'applique quand vous souhaitez modifier l'algorithme qui a été créé, par exemple pour ajouter la journalisation, intercepter des appels de méthode et les suivre, ou dans d'autres buts.

Traduire signifie visiter

Le code que vous générez pour traduire une arborescence d'expressions est une extension de ce que vous avez déjà vu pour visiter tous les nœuds dans une arborescence. Quand vous traduisez une arborescence d'expressions, vous visitez tous les nœuds et, ce faisant, vous générez la nouvelle arborescence. Celle-ci peut contenir des références aux nœuds d'origine ou de nouveaux nœuds que vous avez placés dans l'arborescence.

Nous allons voir cela en action en visitant une arborescence d'expressions et en créant une nouvelle arborescence avec certains nœuds de substitution. Dans cet exemple, nous allons remplacer une constante par une autre dix fois plus grande. Pour le reste, nous laisserons l'arborescence d'expressions intacte. Au lieu de lire la valeur de la constante et de la remplacer par une nouvelle constante, nous allons remplacer le nœud de constante par un nouveau nœud qui effectue la multiplication.

Ici, une fois que vous avez trouvé un nœud de constante, vous créez un nœud de multiplication dont les enfants sont la constante d'origine, et la constante `10` :

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Quand nous remplaçons le nœud d'origine par le substitut, une nouvelle arborescence contenant nos modifications est formée. Nous pouvons vérifier cela en compilant et en exécutant l'arborescence remplacée.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Pour générer une nouvelle arborescence, il faut visiter les nœuds dans l'arborescence existante, créer de nouveaux nœuds et les insérer dans l'arborescence.

Cet exemple montre l'importance du caractère immuable des arborescences d'expressions. Notez que la nouvelle arborescence créée ci-dessus contient une combinaison de nœuds nouvellement créés et de nœuds de l'arborescence existante. Cela ne présente aucun risque, car les nœuds de l'arborescence existante ne peuvent pas être modifiés. Ceci est susceptible d'accroître l'efficacité de la mémoire. Les mêmes nœuds sont utilisables dans toute une arborescence, ou dans plusieurs arborescences d'expressions. Étant donné que les nœuds ne peuvent pas être modifiés, le même nœud peut être réutilisé chaque fois qu'il est nécessaire.

Parcours et exécution d'une addition

Nous allons vérifier cela en générant un deuxième visiteur qui parcourt l'arborescence de nœuds d'addition et calcule le résultat. Pour cela, nous allons apporter quelques modifications au visiteur que nous avons vu jusqu'ici. Dans cette nouvelle version, le visiteur va retourner la somme partielle de l'opération d'addition jusqu'à ce point. Pour une expression constante, il s'agit simplement de la valeur de l'expression constante. Pour une expression d'addition, le résultat est la somme des opérandes gauche et droit, une fois ces arborescences parcourues.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three= Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Il y a pas mal de code ici, mais les concepts sont très abordables. Ce code visite les enfants avec une recherche en profondeur. Quand il rencontre un nœud de constante, le visiteur retourne la valeur de la constante. Une fois que le visiteur a visité les deux enfants, ces enfants auront calculé la somme calculée pour cette sous-arborescence. Le nœud d'addition peut maintenant calculer sa somme. Une fois que tous les nœuds dans l'arborescence d'expressions ont été visités, la somme a été calculée. Vous pouvez suivre l'exécution en exécutant l'exemple dans le débogueur.

Nous allons simplifier le suivi de l'analyse des nœuds et du calcul de la somme en parcourant l'arborescence. Voici une version mise à jour de la méthode Aggregate qui comprend des informations de suivi :

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Son exécution sur la même expression génère la sortie suivante :

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Suivez la sortie et suivez le code ci-dessus. Vous devriez pouvoir déterminer comment le code visite chaque nœud et calcule la somme à mesure qu'il parcourt l'arborescence et trouve la somme.

À présent, jetons un œil à une autre exécution, avec l'expression donnée par `sum1` :

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

Voici la sortie de l'examen de cette expression :

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Bien que la réponse finale soit la même, le parcours d'arborescence est complètement différent. Les nœuds sont parcourus dans un ordre différent, car l'arborescence a été construite avec différentes opérations survenant en premier.

En savoir plus

Cet exemple montre un petit sous-ensemble du code que vous créeriez pour parcourir et interpréter les algorithmes représentés par une arborescence d'expressions. Pour obtenir une description complète de tout le travail nécessaire pour générer une bibliothèque à usage général qui traduit des arborescences d'expressions dans un autre langage, consultez cette série de Matt Warren. Elle décrit en détail comment traduire le code que vous pourriez rencontrer dans une arborescence d'expressions.

J'espère que cet article aura mis en évidence toute la puissance des arborescences d'expressions. Vous pouvez examiner un ensemble de code, y apporter les modifications souhaitées et exécuter la version modifiée. Les arborescences d'expressions étant immuables, vous pouvez créer des arborescences à l'aide de composants d'arborescences existantes. Cela réduit la quantité de mémoire nécessaire pour créer des arborescences d'expressions.

Suivant -- Récapitulatif

Récapitulatif concernant les arborescences d'expressions

18/03/2020 • 2 minutes to read • Edit Online

Précédent -- Traduction d'expressions

Dans cette série, vous avez vu comment utiliser des *arborescences d'expressions* pour créer des programmes dynamiques qui interprètent du code en tant que données et génèrent de nouvelles fonctionnalités en fonction de ce code.

Vous pouvez examiner des arborescences d'expressions pour comprendre l'intention d'un algorithme. Vous pouvez non seulement examiner ce code, mais aussi créer de nouvelles arborescences d'expressions qui représentent des versions modifiées du code d'origine.

Vous pouvez également utiliser des arborescences d'expressions pour examiner un algorithme, et traduire celui-ci dans un autre langage ou environnement.

Limites

Certains éléments de langage C# récents ne se traduisent pas correctement en arborescences d'expressions. Les arborescences d'expressions ne peuvent pas contenir d'expressions `await` ou d'expressions lambda `async`. La plupart des fonctionnalités ajoutées dans la version 6 de C# n'apparaissent pas exactement telles qu'écrites dans les arborescences d'expressions. Au lieu de cela, les nouvelles fonctionnalités sont exposées dans les arborescences d'expressions dans la syntaxe antérieure équivalente. Cette limitation n'est peut-être pas aussi importante que vous pourriez le penser. En fait, elle signifie que votre code qui interprète des arborescences d'expressions continuera probablement de fonctionner de la même manière lors de l'introduction de nouvelles fonctionnalités de langage.

Même avec ces limitations, les arborescences d'expressions vous permettent de créer des algorithmes dynamiques qui reposent sur l'interprétation et la modification de code représenté sous la forme d'une structure de données. C'est un outil puissant et l'une des fonctionnalités de l'écosystème .NET qui permet aux bibliothèques enrichies telles qu'Entity Framework d'accomplir leur travail.

Interopérabilité (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

L'interopérabilité vous permet de préserver et de tirer parti d'investissements existants en code non managé. Le code qui s'exécute sous le contrôle du common language runtime (CLR) est appelé *code managé*, et le code qui s'exécute en dehors du CLR est appelé *code non managé*. COM, COM+, les composants C++, les composants ActiveX et l'API Microsoft Windows sont des exemples de code non managé.

.NET permet l'interopérabilité avec du code non managé via des services d'appel de code non managé, l' System.Runtime.InteropServices espace de noms, l'interopérabilité C++ et l'interopérabilité COM (COM Interop).

Dans cette section

Vue d'ensemble de l'interopérabilité
Décrit les méthodes qui permettent l'interopérabilité entre le code managé et le code non managé en C#.

Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#
Décrit les fonctionnalités introduites dans Visual C# pour faciliter la programmation Office.

Comment utiliser des propriétés indexées dans la programmation COM Interop
Explique comment utiliser des propriétés indexées pour accéder aux propriétés COM qui ont des paramètres.

Comment utiliser l'appel de code non managé pour lire un fichier WAV
Explique comment utiliser des services d'appel de code non managé pour lire un fichier son .wav sur le système d'exploitation Windows.

Procédure pas à pas : programmation Office
Montre comment créer un classeur Excel et un document Word qui contient un lien vers le classeur.

Exemple de classe COM
Montre comment exposer une classe C# en tant qu'objet COM.

Spécification du langage C#

Pour plus d'informations, consultez Concepts de base dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Marshal.ReleaseComObject
- Guide de programmation C#
- Interopération avec du code non managé
- Procédure pas à pas : programmation Office

Documenter votre code avec des commentaires XML

02/11/2020 • 37 minutes to read • Edit Online

Les commentaires de documentation XML sont un genre particulier de commentaire, ajouté au-dessus de la définition d'un type ou d'un membre défini par l'utilisateur. Ils sont spéciaux, car ils peuvent être traités par le compilateur pour générer un fichier de documentation XML au moment de la compilation. Le fichier XML généré par le compilateur peut être distribué avec votre assembly .NET afin que Visual Studio et d'autres IDE puissent utiliser IntelliSense pour afficher des informations rapides sur les types ou les membres. De plus, le fichier XML peut être exécuté par l'intermédiaire d'outils tels que DocFX et Sandcastle pour générer des sites web de références d'API.

Les commentaires de documentation XML, comme tous les autres commentaires, sont ignorés par le compilateur.

Vous pouvez générer le fichier XML au moment de la compilation en procédant comme suit :

- Si vous développez une application avec .NET Core à partir de la ligne de commande, vous pouvez ajouter un `GenerateDocumentationFile` élément à la `<PropertyGroup>` section de votre fichier projet. csproj. Vous pouvez également spécifier le chemin d'accès au fichier de documentation directement à l'aide de l' `DocumentationFile` élément. L'exemple suivant génère un fichier XML dans le répertoire du projet avec le même nom de fichier racine que l'assembly :

  ```
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  ```

  est équivalente à celle-ci :

  ```
  <DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
  ```

- Si vous développez une application à l'aide de Visual Studio, cliquez avec le bouton droit sur le projet, puis sélectionnez **Propriétés**. Dans la boîte de dialogue des propriétés, sélectionnez l'onglet **Générer**, puis cochez **Fichier de documentation XML**. Vous pouvez également changer l'emplacement dans lequel le compilateur écrit le fichier.

- Si vous compilez une application .NET à partir de la ligne de commande, ajoutez l' option de compilateur-doc lors de la compilation.

Les commentaires de documentation XML utilisent des barres obliques triples (`///`) et le corps d'un commentaire au format XML. Par exemple :

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{
}
```

Procédure pas à pas

Passons en revue la documentation d'une bibliothèque mathématique très basique pour permettre aux nouveaux développeurs de comprendre et de contribuer facilement et aux développeurs tiers de les utiliser.

Voici le code de la bibliothèque mathématique simple :

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

L'exemple de bibliothèque prend en charge quatre opérations arithmétiques majeures (`add` ,, `subtract` `multiply` et `divide`) sur les `int` types de `double` données et.

Vous souhaitez maintenant être en mesure de créer un document de référence d'API à partir de votre code pour les développeurs tiers qui utilisent votre bibliothèque, mais n'ont pas accès au code source. Comme nous l'avons déjà mentionné, vous pouvez pour cela utiliser les balises de documentation XML. Vous allez maintenant découvrir les balises XML standard prises en charge par le compilateur C#.

<summary>

La balise `<summary>` ajoute des informations succinctes relatives à un type ou un membre. Je vais expliquer son utilisation en l'ajoutant à la définition de la classe `Math` et à la première méthode `Add`. N'hésitez pas à l'appliquer au reste de votre code.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

La `<summary>` balise est importante et nous vous recommandons de l'inclure car son contenu est la principale source d'informations sur les types ou les membres dans IntelliSense ou un document de référence sur les API.

<remarks>

La balise `<remarks>` complète les informations relatives aux types ou aux membres fournies par la balise `<summary>`. Dans cet exemple, vous allez simplement l'ajouter à la classe.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{
}
```

<returns>

La balise `<returns>` décrit la valeur de retour d'une déclaration de méthode. Comme auparavant, l'exemple suivant illustre la balise `<returns>` sur la première méthode `Add`. Vous pouvez effectuer la même opération sur d'autres méthodes.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

<value>

La balise `<value>` est similaire à la balise `<returns>`, excepté que vous l'utilisez pour les propriétés. En supposant que votre bibliothèque `Math` a une propriété statique appelée `PI`, voici comment utiliser cette balise :

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

<example>

La balise `<example>` permet d'inclure un exemple de votre documentation XML. Cela implique l'utilisation de la balise enfant `<code>`.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

La balise `code` conserve les sauts de ligne et la mise en retrait pour les exemples plus longs.

<para>

La balise `<para>` permet de mettre en forme le contenu de sa balise parente. `<para>` est généralement utilisée dans une balise, telle que `<remarks>` ou `<returns>`, pour diviser du texte en paragraphes. Vous pouvez mettre en forme le contenu de la balise `<remarks>` pour la définition de votre classe.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
}
```

<c>

Toujours concernant la mise en forme, vous utilisez la balise `<c>` pour le marquage d'une partie du texte comme code. Elle est semblable à la balise `<code>`, mais inline. Elle est utile quand vous voulez afficher un exemple de code rapide comme partie du contenu d'une balise. Mettons à jour la documentation de la classe `Math`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
}
```

<exception>

En utilisant la balise `<exception>`, vous informez vos développeurs qu'une méthode peut lever des exceptions spécifiques. Si vous examinez votre bibliothèque `Math`, vous pouvez voir que les deux méthodes `Add` lèvent une exception si une certaine condition est remplie. Il est toutefois moins évident de voir que la méthode `Divide` utilisée avec un entier lève également une exception si le paramètre `b` est égal à zéro. Ajoutez maintenant une documentation d'exception à cette méthode.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
```

```
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Divides an integer by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two integers.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    /// <summary>
    /// Divides a double by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two doubles.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

L'attribut `cref` référence une exception qui est disponible à partir de l'environnement de compilation actuel. Il peut s'agir de tout type défini dans le projet ou dans un assembly référencé. Le compilateur émet un avertissement si sa valeur ne peut pas être résolue.

<see>

La balise `<see>` vous permet de créer un lien interactif vers une page de documentation pour un autre élément de code. Dans notre prochain exemple, nous allons créer un lien interactif entre les deux méthodes `Add`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

`cref` est un attribut **obligatoire** qui représente une référence à un type ou à ses membres et qui est disponible à partir de l'environnement de compilation actuel. Il peut s'agir de tout type défini dans le projet ou dans un assembly référencé.

<seealso>

Vous pouvez utiliser la balise `<seealso>` de la même façon que la balise `<see>`. La seule différence est que son contenu est généralement placé dans une section "Voir aussi". Ici, nous allons ajouter une balise `seealso` sous la méthode `Add` utilisée avec un entier pour référencer d'autres méthodes de la classe qui acceptent des paramètres entiers :

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

L'attribut `cref` représente une référence à un type ou à ses membres et est disponible à partir de l'environnement de compilation actuel. Il peut s'agir de tout type défini dans le projet ou dans un assembly référencé.

<param>

La balise `<param>` permet de décrire les paramètres d'une méthode. Voici un exemple de la méthode double `Add` : le paramètre que la balise décrit est spécifié dans l'attribut **Required** `name` .

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparam>

Vous utilisez la balise `<typeparam>` exactement comme la balise `<param>`, mais pour permettre aux déclarations de types ou de méthodes génériques de décrire un paramètre générique. Ajoutez une méthode générique rapide à votre classe `Math` pour vérifier si une quantité est supérieure à une autre.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<paramref>

Alors que vous décrivez ce que fait une méthode dans ce qui pourrait être une balise `<summary>`, vous souhaiterez peut-être créer une référence à un paramètre. La balise `<paramref>` est idéale pour cette tâche précise. Mettons à jour le récapitulatif de notre méthode `Add` double. À l'instar de la `<param>` balise, le nom du paramètre est spécifié

dans l'attribut **Required** `name` .

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparamref>

Vous utilisez la balise `<typeparamref>` exactement comme la balise `<paramref>`, mais pour permettre aux déclarations de types ou de méthodes génériques de décrire un paramètre générique. Vous pouvez utiliser la même méthode générique que celle créée précédemment.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<list>

Vous utilisez la `<list>` balise pour mettre en forme les informations de documentation sous la forme d'une liste

triée, d'une liste non triée ou d'une table. Créez une liste non triée de toutes les opérations mathématiques prises en charge par votre bibliothèque `Math`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{
}
```

Vous pouvez créer une liste triée ou un tableau en remplaçant l'attribut `type` par `number` ou `table`, respectivement.

<inheritdoc>

Vous pouvez utiliser la `<inheritdoc>` balise pour hériter des commentaires XML des classes de base, des interfaces et des méthodes similaires. Cela élimine la copie et le collage indésirables de commentaires XML en double et conserve automatiquement les commentaires XML synchronisés.

```
/*
    The IMath interface
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// This is the IMath interface.
/// </summary>
public interface IMath
{
}

/// <inheritdoc/>
public class Math : IMath
{
}
```

Assemblage

Si vous avez suivi ce didacticiel et appliqué les balises à votre code au besoin, votre code doit maintenant

ressembler à ce qui suit :

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
```

```
    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Add(4.5, 5.4);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <summary>
    /// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The difference between two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Subtract(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <summary>
    /// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
    /// </summary>
    /// <returns>
    /// The difference between two doubles.
    /// </returns>
```

```
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Subtract(4.5, 5.4);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <summary>
    /// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Multiply(4, 5);
    /// if (c > 100)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <summary>
    /// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Multiply(4.5, 5.4);
    /// if (c > 100.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
```

```
    /// <param name="b">A double precision number.</param>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <summary>
    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <summary>
    /// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Divide(4.5, 5.4);
    /// if (c > 1.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(int, int)"/> to divide integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

À partir de votre code, vous pouvez générer un site web de documentation détaillée complet avec des références croisées interactives. Vous êtes maintenant confronté à un autre problème : votre code est devenu difficile à lire.

Avec toutes ces informations à parcourir, cela va être un cauchemar pour les développeurs qui veulent contribuer à ce code. Heureusement, il existe une balise XML qui peut vous aider à gérer ce problème :

<include>

La balise `<include>` vous permet de faire référence à des commentaires se trouvant dans un fichier XML distinct et décrivant les types et les membres de votre code source au lieu de placer des commentaires de documentation directement dans votre fichier de code source.

Vous allez maintenant déplacer toutes vos balises XML dans un fichier XML distinct nommé `docs.xml`. Vous pouvez nommer ce fichier comme vous le souhaitez.

```
<docs>
    <members name="math">
        <Math>
            <summary>
            The main <c>Math</c> class.
            Contains all methods for performing basic math functions.
            </summary>
            <remarks>
            <para>This class can add, subtract, multiply and divide.</para>
            <para>These operations can be performed on both integers and doubles.</para>
            </remarks>
        </Math>
        <AddInt>
            <summary>
            Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The sum of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Add(4, 5);
            if (c > 10)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.OverflowException">Thrown when one parameter is max
            and the other is greater than 0.</exception>
            See <see cref="Math.Add(double, double)"/> to add doubles.
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </AddInt>
        <AddDouble>
            <summary>
            Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The sum of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Add(4.5, 5.4);
            if (c > 10)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
```

```
            <exception cref="System.OverflowException">Thrown when one parameter is max
            and the other is greater than 0.</exception>
            See <see cref="Math.Add(int, int)"/> to add integers.
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </AddDouble>
        <SubtractInt>
            <summary>
            Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
            </summary>
            <returns>
            The difference between two integers.
            </returns>
            <example>
            <code>
            int c = Math.Subtract(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </SubtractInt>
        <SubtractDouble>
            <summary>
            Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
            </summary>
            <returns>
            The difference between two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Subtract(4.5, 5.4);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Subtract(int, int)"/> to subtract integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </SubtractDouble>
        <MultiplyInt>
            <summary>
            Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Multiply(4, 5);
            if (c > 100)
            {
                Console.WriteLine(c);
```

```
            }
            </code>
            </example>
            See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </MultiplyInt>
        <MultiplyDouble>
            <summary>
            Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Multiply(4.5, 5.4);
            if (c > 100.0)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Multiply(int, int)"/> to multiply integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </MultiplyDouble>
        <DivideInt>
            <summary>
            Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Divide(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(double, double)"/> to divide doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <param name="a">An integer dividend.</param>
            <param name="b">An integer divisor.</param>
        </DivideInt>
        <DivideDouble>
            <summary>
            Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two doubles.
            </returns>
            <example>
            <code>
```

```
            double c = Math.Divide(4.5, 5.4);
            if (c > 1.0)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(int, int)"/> to divide integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <param name="a">A double precision dividend.</param>
            <param name="b">A double precision divisor.</param>
        </DivideDouble>
    </members>
</docs>
```

Dans le code XML ci-dessus, les commentaires de documentation de chaque membre apparaissent directement dans une balise nommée en fonction de ce qu'il fait. Vous pouvez choisir votre propre stratégie. Maintenant que vous avez vos commentaires XML dans un fichier distinct, voyons comment votre code peut être rendu plus lisible à l'aide de la balise `<include>` :

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'/>
public class Math
{
    // Adds two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'/>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'/>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'/>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'/>
    public static double Subtract(double a, double b)
    {
```

```
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'/>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'/>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'/>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'/>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

Et voilà : notre code est à nouveau lisible, et aucune information de documentation n'a été perdue.

L'attribut `file` représente le nom du fichier XML contenant la documentation.

L'attribut `path` représente une requête XPath au `tag name` présent dans le `file` spécifié.

L'attribut `name` représente le spécificateur de nom dans la balise qui précède les commentaires.

L' `id` attribut, qui peut être utilisé à la place de `name` , représente l'ID de la balise qui précède les commentaires.

Balises définies par l'utilisateur

Toutes les balises décrites ci-dessus représentent celles qui sont reconnues par le compilateur C#. Toutefois, un utilisateur est libre de définir ses propres balises. Les outils tels que Sandcastle apportent la prise en charge de balises supplémentaires comme <event> et <note> , et prennent même en charge la documentation des espaces de noms. Des outils de génération de documentation personnalisés ou internes peuvent également être utilisés avec les balises standard, et plusieurs formats de sortie, du format HTML au format PDF, peuvent être pris en charge.

Recommandations

La documentation du code est recommandée pour de nombreuses raisons. Voici quelques bonnes pratiques, des scénarios de cas d'usage généraux et des éléments que vous devez connaître quand vous utilisez des balises de documentation XML dans votre code C#.

- Par souci de cohérence, tous les types visibles publiquement et leurs membres doivent être documentés. Si vous devez le faire, faites-le complètement.
- Les membres privés peuvent également être documentés à l'aide de commentaires XML. Toutefois, cela expose le fonctionnement interne (potentiellement confidentiel) de votre bibliothèque.
- Au minimum, les types et leurs membres doivent avoir une balise `<summary>`, car son contenu est nécessaire

pour IntelliSense.

- Le texte de la documentation doit être écrit à l'aide de phrases complètes se terminant par un point.
- Les classes partielles sont entièrement prises en charge, et les informations de documentation seront concaténées dans une seule entrée pour ce type.
- Le compilateur vérifie la syntaxe des `<exception>` `<include>` balises,,,, `<param>` `<see>` `<seealso>` et `<typeparam>` .
- Le compilateur valide les paramètres qui contiennent des chemins de fichiers et des références à d'autres parties du code.

Voir aussi

- Commentaires de documentation XML (Guide de programmation C#)
- Balises recommandées pour les commentaires de documentation (Guide de programmation C#)

Gestion de versions en C#

24/04/2020 • 11 minutes to read • Edit Online

Dans ce didacticiel, vous allez apprendre la signification de la gestion de version .NET. Vous découvrirez également les facteurs à prendre en compte dans le cadre du contrôle de version de votre bibliothèque et de sa mise à niveau vers une nouvelle version.

Création de bibliothèques

En tant que développeur ayant créé des bibliothèques .NET pour une utilisation publique, vous avez probablement été confronté à des situations où vous deviez déployer de nouvelles mises à jour. Votre manière de procéder est très importante, car vous devez assurer une transition fluide du code existant vers la nouvelle version de votre bibliothèque. Voici quelques points à prendre en compte lors de la création d'une version :

Gestion sémantique des versions

La gestion sémantique de version (SemVer en abrégé) est une convention d'affectation de noms appliquée à des versions de votre bibliothèque pour indiquer des événements jalons spécifiques. Dans l'idéal, les informations de version ajoutées à votre bibliothèque doivent aider les développeurs à déterminer la compatibilité avec les projets qui utilisent des versions antérieures de cette même bibliothèque.

L'approche la plus simple de SemVer est le format à 3 composants `MAJOR.MINOR.PATCH`, où :

- `MAJOR` est incrémenté quand vous apportez des changements d'API incompatibles
- `MINOR` est incrémenté quand vous ajoutez des fonctionnalités à compatibilité descendante
- `PATCH` est incrémenté quand vous apportez des correctifs de bogues à compatibilité descendante

Il existe également des moyens de spécifier d'autres scénarios, tels que des préversions, lors de l'application des informations de version à votre bibliothèque .NET.

Compatibilité descendante

Quand vous publiez de nouvelles versions de votre bibliothèque, l'une de vos principales préoccupations concerne la compatibilité descendante avec les versions précédentes. Une nouvelle version de la bibliothèque est compatible au format source avec une version précédente si le code qui dépend de la version précédente est utilisable avec la nouvelle version une fois qu'il est recompilé. Une nouvelle version de la bibliothèque est compatible au format binaire si une application qui dépendait de l'ancienne version est utilisable avec la nouvelle version sans recompilation.

Voici quelques éléments à prendre en compte quand vous tentez de maintenir la compatibilité descendante avec d'anciennes versions de votre bibliothèque :

- Méthodes virtuelles : quand vous rendez une méthode virtuelle non virtuelle dans la nouvelle version, cela signifie que les projets qui substituent cette méthode doivent être mis à jour. Il s'agit là d'une modification avec rupture importante qui est vivement déconseillée.
- Signatures de méthode : Lors de la mise à jour d'un comportement de méthode vous oblige à changer sa signature ainsi, vous devriez plutôt créer une surcharge de sorte que le code appelant dans cette méthode fonctionnera toujours. Vous pouvez toujours manipuler l'ancienne signature de méthode pour appeler la nouvelle signature de méthode afin que l'implémentation reste cohérente.
- Attribut obsolète : vous pouvez utiliser cet attribut dans votre code pour spécifier des classes ou membres de classe dépréciés et susceptibles d'être supprimés dans les prochaines versions. Cela garantit que les développeurs utilisant votre bibliothèque sont mieux préparés pour les modifications avec rupture.

- Arguments de méthode facultatifs : quand vous rendez obligatoires des arguments de méthode auparavant facultatifs ou modifiez leur valeur par défaut, tout le code qui ne fournit pas ces arguments doit être mis à jour.

> **NOTE**
> Rendre les arguments obligatoires facultatifs devrait avoir très peu d'effet, surtout si cela ne change pas le comportement de la méthode.

Plus il est facile pour vos utilisateurs d'effectuer la mise à niveau vers la nouvelle version de votre bibliothèque, plus il est probable qu'ils le feront rapidement.

Fichier de configuration de l'application

En tant que développeur .NET, il est très probable que vous ayez rencontré le fichier `app.config`, présent dans la plupart des types de projets. Ce simple fichier de configuration peut s'avérer très utile pour améliorer le déploiement de nouvelles mises à jour. Vous devez généralement concevoir vos bibliothèques de telle sorte que les `app.config` informations qui sont susceptibles de changer régulièrement sont stockées dans le fichier, de cette façon lorsque ces informations sont mises à jour, le fichier config des anciennes versions doit juste être remplacé par la nouvelle sans avoir besoin de recomplification de la bibliothèque.

Utilisation des bibliothèques

En tant que développeur qui utilise les bibliothèques .NET générées par d'autres développeurs, vous savez probablement qu'une nouvelle version d'une bibliothèque peut ne pas être totalement compatible avec votre projet et vous pouvez souvent être obligé de mettre à jour votre code pour vous adapter à ces modifications.

Heureusement pour vous, C et l'écosystème .NET est livré avec des fonctionnalités et des techniques qui nous permettent de mettre à jour facilement notre application pour travailler avec de nouvelles versions de bibliothèques qui pourraient introduire des changements de rupture.

Redirection des liaisons d'assembly

Vous pouvez utiliser le fichier *app.config* pour mettre à jour la version d'une bibliothèque que votre application utilise. En ajoutant ce qu'on appelle une *redirection de liaison,*vous pouvez utiliser la nouvelle version de la bibliothèque sans avoir à recomposer votre application. L'exemple suivant montre comment vous mettre à jour le `1.0.1` fichier *app.config* de votre application pour utiliser la version patch au lieu de `ReferencedLibrary` la `1.0.0` version avec laquelle il a été initialement compilé.

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **NOTE**
> Cette approche fonctionne uniquement si la nouvelle version de `ReferencedLibrary` est compatible au format binaire avec votre application. Consultez la section Compatibilité descendante ci-dessus pour connaître les changements à rechercher lors de la détermination de la compatibilité.

new

Vous utilisez le modificateur `new` pour masquer les membres hérités d'une classe de base. Il s'agit d'une façon pour les classes dérivées de pouvoir répondre aux mises à jour dans les classes de base.

Prenons l'exemple suivant :

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Sortie

```
A base method
A derived method
```

Dans l'exemple ci-dessus, vous pouvez voir comment `DerivedClass` masque la méthode `MyMethod` présente dans `BaseClass`. Cela signifie que, quand une classe de base dans la nouvelle version d'une bibliothèque ajoute un membre qui existe déjà dans votre classe dérivée, vous pouvez simplement utiliser le modificateur `new` sur le membre de la classe dérivée pour masquer le membre de la classe de base.

Quand aucun modificateur `new` n'est spécifié, une classe dérivée masque par défaut des membres en conflit dans une classe de base ; même si un avertissement du compilateur est généré, le code est toujours compilé. Cela signifie que le fait d'ajouter simplement de nouveaux membres à une classe existante rend cette nouvelle version de votre bibliothèque compatible au format source et binaire avec le code qui en dépend.

override

Le modificateur `override` indique qu'une implémentation dérivée étend l'implémentation d'un membre de classe de base au lieu de la masquer. Le modificateur `virtual` doit être appliqué au membre de classe de base.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Sortie

```
Base Method One: Method One
Derived Method One: Derived Method One
```

Le modificateur `override` est évalué au moment de la compilation et génère une erreur s'il ne trouve pas un membre virtuel à substituer.

Votre connaissance des techniques discutées et votre compréhension des situations dans lesquelles les utiliser, contribueront grandement à faciliter la transition entre les versions d'une bibliothèque.

Guides pratiques (C#)

02/11/2020 • 7 minutes to read • Edit Online

Dans la section How to du guide C#, vous trouverez des réponses rapides aux questions les plus fréquentes. Dans certains cas, les articles peuvent figurer dans plusieurs sections. Nous avons souhaité les rendre plus faciles à trouver via plusieurs voies de recherche.

Concepts C# généraux

Il existe plusieurs trucs et astuces qui sont des pratiques courantes pour les développeurs C# :

- Initialiser des objets à l'aide d'un initialiseur d'objet.
- Découvrir les différences entre le passage d'un struct et d'une classe à une méthode.
- Utiliser la surcharge d'opérateur.
- Implémenter et appeler une méthode d'extension personnalisée.
- Même les programmeurs C# peuvent souhaiter utiliser l' `My` espace de noms de Visual Basic.
- Créer une méthode pour un type `enum` à l'aide des méthodes d'extension.

Membres de classe et de struct

Vous créez des classes et des structs pour implémenter votre programme. Ces techniques sont couramment utilisées lors de l'écriture de classes ou de structs.

- Déclarer des propriétés implémentées automatiquement.
- Déclarer et utiliser des propriétés en lecture-écriture.
- Définir des constantes.
- Remplacer la méthode `ToString` pour fournir la sortie de chaîne.
- Définir des propriétés abstraites.
- Utiliser les fonctionnalités de la documentation XML pour documenter votre code.
- Implémenter de manière explicite des membres d'interface pour conserver votre interface publique concise.
- Implémenter de manière explicite des membres de deux interfaces.

Utilisation des collections

Ces articles vous permettent d'utiliser des collections de données.

- Initialiser un dictionnaire avec un initialiseur de collection.

Utilisation des chaînes

Les chaînes constituent le type de données fondamental utilisé pour afficher ou manipuler du texte. Ces articles présentent les pratiques courantes avec des chaînes.

- Comparer des chaînes.
- Modifier le contenu d'une chaîne.
- Déterminer si une chaîne représente un nombre.
- Utiliser `String.Split` pour séparer les chaînes.
- Combiner plusieurs chaînes en une seule.
- Rechercher du texte dans une chaîne.

Conversion entre types

Vous aurez peut-être besoin de convertir un objet en un autre type.

- Déterminer si une chaîne représente un nombre.
- Convertir entre des chaînes qui représentent des nombres hexadécimaux et le nombre.
- Convertir une chaîne en `DateTime`.
- Convertir un tableau d'octets en int.
- Convertir une chaîne en nombre .
- Utiliser les critères spéciaux et les opérateurs `as` et `is` pour caster sans risque en un autre type.
- Définir des conversions de type personnalisé.
- Déterminer si un type est un type valeur nullable.
- Convertir entre des types valeur nullable et non nullable.

Comparaisons d'égalité et de classement

Vous pouvez créer des types qui définissent leurs propres règles pour l'égalité ou qui définissent un classement naturel entre les objets de ce type.

- Tester l'égalité basée sur les références.
- Définir une égalité basée sur les valeurs pour un type.

Gestion des exceptions

Les programmes .NET signalent que des méthodes n'ont pas réussi à effectuer leur travail en levant des exceptions. Dans ces articles, vous allez apprendre à travailler avec les exceptions.

- Gérer les exceptions avec `try` et `catch`.
- Nettoyer les ressources avec les clauses `finally`.
- Récupérer suite à des exceptions non-CLS (Common Language Specification).

Délégués et événements

Les délégués et les événements fournissent une fonctionnalité pour les stratégies qui implique des blocs de code faiblement couplés.

- Déclarer, instancier et utiliser des délégués.
- Combiner des délégués multicast.

Les événements offrent un mécanisme permettant de publier des notifications ou de s'y abonner.

- S'abonner à des événements et s'en désabonner.
- Implémenter des événements déclarés dans des interfaces.
- Se conformer aux instructions .net lorsque votre code publie des événements.
- Déclencher des événements définis dans les classes de base de classes dérivées.
- Implémenter des accesseurs d'événement personnalisés.

Pratiques LINQ

LINQ vous permet d'écrire du code pour interroger une source de données qui prend en charge le modèle d'expression de requête LINQ. Ces articles vous permettent de comprendre le modèle et d'utiliser différentes sources de données.

- Interroger une collection.

- Utiliser des expressions lambda dans une requête.
- Utiliser `var` dans des expressions de requête.
- Retourner des sous-ensembles de propriétés d'éléments d'une requête.
- Écrire des requêtes avec un filtrage complexe.
- Trier les éléments d'une source de données.
- Trier les éléments sur plusieurs clés.
- Contrôler le type d'une projection.
- Compter les occurrences d'une valeur dans une séquence source.
- Calculer des valeurs intermédiaires.
- Fusionner des données de plusieurs sources.
- Trouver la différence définie entre deux séquences.
- Déboguer des résultats de requête vides.
- Ajouter des méthodes personnalisées dans des requêtes LINQ.

Traitement de plusieurs threads et traitement asynchrone

Les programmes récents utilisent souvent des opérations asynchrones. Les articles suivants vous apprennent à utiliser ces techniques.

- Améliorer les performances asynchrones avec `System.Threading.Tasks.Task.WhenAll`.
- Effectuer plusieurs requêtes web en parallèle en utilisant `async` et `await`.
- Utiliser un pool de threads.

Arguments de ligne de commande dans votre programme

En règle générale, les programmes en C# ont des arguments de ligne de commande. Les articles suivants vous apprennent à accéder et à traiter ces arguments de ligne de commande.

- Récupérer tous les arguments de ligne de commande avec `for`.

Comment séparer des chaînes à l'aide de String. Split en C#

16/12/2020 • 4 minutes to read • Edit Online

La méthode String.Split crée un tableau de sous-chaînes en fractionnant la chaîne d'entrée en fonction d'un ou plusieurs délimiteurs. Cette méthode est souvent le moyen le plus simple de séparer une chaîne sur les limites de mots. Il est également utilisé pour fractionner des chaînes sur d'autres caractères ou chaînes spécifiques.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Le code suivant fractionne une expression commune en un tableau de chaînes pour chaque mot.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Chaque instance d'un caractère de séparation génère une valeur dans le tableau retourné. Les caractères de séparation consécutifs produisent une chaîne vide comme valeur dans le tableau retourné. Vous pouvez voir comment une chaîne vide est créée dans l'exemple suivant, qui utilise le caractère espace comme séparateur.

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Ce comportement facilite l'utilisation de formats tels que des fichiers de valeurs séparées par des virgules (CSV) représentant des données tabulaires. Les virgules consécutives représentent une colonne vide.

Vous pouvez passer un paramètre StringSplitOptions.RemoveEmptyEntries facultatif pour exclure toutes les chaînes vides dans le tableau retourné. Pour un traitement plus complexe de la collection retournée, vous pouvez utiliser LINQ pour manipuler la séquence de résultat.

String.Split peut utiliser plusieurs caractères de séparation. L'exemple suivant utilise des espaces, des virgules, des points, des deux-points et des tabulations comme séparateurs de caractères, qui sont passés à Split dans un tableau. La boucle en bas du code affiche chacun des mots dans le tableau retourné.

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Les instances consécutives d'un séparateur produisent une chaîne vide dans le tableau de sortie :

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

String.Split peut accepter un tableau de chaînes (séquences de caractères, à la place de caractères uniques, qui agissent comme séparateurs lors de l'analyse de la chaîne cible).

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

Voir aussi

- Extraire des éléments d'une chaîne
- Guide de programmation C#
- Chaînes
- Expressions régulières .NET

Comment concaténer plusieurs chaînes (Guide C#)

02/11/2020 • 6 minutes to read • Edit Online

La *concaténation* consiste à ajouter une chaîne à la fin d'une autre chaîne. Vous concaténez les chaînes à l'aide de l'opérateur `+`. Pour les littéraux de chaîne et les constantes de chaîne, la concaténation se produit au moment de la compilation ; aucune concaténation ne se produit au moment de l'exécution. Pour les variables de chaîne, la concaténation se produit uniquement au moment de l'exécution.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

L'exemple suivant utilise la concaténation pour diviser un long littéral de chaîne en plus petites chaînes pour améliorer la lisibilité du code source. Les diverses parties sont concaténées en une chaîne unique lors de la compilation. Il n'y a aucune incidence sur les performances d'exécution, quel que soit le nombre de chaînes impliquées.

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a " +
"query can only be represented as a string without compile-time type checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

Pour concaténer des variables de chaîne, vous pouvez utiliser les `+` `+=` opérateurs ou, l'interpolation de chaîne ou les String.Format String.Concat méthodes, String.Join ou StringBuilder.Append . L'opérateur `+` est facile à utiliser et convient au code intuitif. Même si vous utilisez plusieurs opérateurs `+` dans une instruction, le contenu de la chaîne est copié une seule fois. Le code suivant montre des exemples d'utilisation des opérateurs `+` et `+=` pour concaténer des chaînes :

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

Dans certaines expressions, il est plus facile de concaténer des chaînes à l'aide de l'interpolation de chaînes, comme le montre le code suivant :

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> **NOTE**
> Dans les opérations de concaténation de chaîne, le compilateur C# traite une chaîne null de la même manière qu'une chaîne vide.

Une autre méthode permettant de concaténer des chaînes est String.Format. Cette méthode fonctionne bien quand vous créez une chaîne à partir d'un petit nombre de chaînes de composant.

Dans d'autres cas, vous pouvez combiner des chaînes dans une boucle où vous ne connaissez pas le nombre de chaînes sources que vous combinez et le nombre réel de chaînes sources peut être élevé. La classe StringBuilder a été conçue pour ces scénarios. Le code suivant utilise la méthode Append de la classe StringBuilder pour concaténer des chaînes.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

Vous pouvez en savoir plus sur les raisons pour choisir la concaténation de chaînes ou la `StringBuilder` classe.

Une autre option permettant de joindre les chaînes d'une collection consiste à utiliser la méthode String.Concat. Utilisez String.Join la méthode si les chaînes sources doivent être séparées par un délimiteur. Le code suivant combine un tableau de mots en utilisant ces deux méthodes :

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

Enfin, vous pouvez utiliser LINQ et la méthode Enumerable.Aggregate pour joindre les chaînes d'une collection. Cette méthode combine les chaînes sources en utilisant une expression lambda. L'expression lambda effectue le travail d'ajouter chaque chaîne à l'accumulation existante. L'exemple suivant combine un tableau de mots en ajoutant un espace entre chaque mot du tableau :

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

Voir aussi

- String
- StringBuilder
- Guide de programmation C#
- Chaînes

Comment rechercher des chaînes

02/11/2020 • 7 minutes to read • Edit Online

Vous pouvez utiliser deux stratégies principales pour rechercher du texte dans des chaînes. Les méthodes de la classe String recherchent un texte spécifique. Les expressions régulières recherchent des modèles dans du texte.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Le type string, qui est un alias de la classe System.String, propose un certain nombre de méthodes utiles pour rechercher le contenu d'une chaîne. En voici quelques-unes : Contains, StartsWith, EndsWith, IndexOf, LastIndexOf. La classe System.Text.RegularExpressions.Regex fournit un vocabulaire étendu pour rechercher des modèles dans du texte. Dans cet article, vous découvrez ces techniques et comment choisir la meilleure méthode pour vos besoins.

Une chaîne contient-elle un certain texte ?

Les String.Contains String.StartsWith méthodes, et String.EndsWith recherchent un texte spécifique dans une chaîne. L'exemple suivant illustre chacune de ces méthodes et une variation qui utilise une recherche ne respectant pas la casse :

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

L'exemple précédent montre un point important de l'utilisation de ces méthodes. Par défaut, les recherches **respectent la casse**. Vous utilisez la StringComparison.CurrentCultureIgnoreCase valeur d'énumération pour spécifier une recherche qui ne respecte pas la casse.

Où le texte recherché se trouve-t-il dans une chaîne ?

Les méthodes IndexOf et LastIndexOf recherchent aussi du texte dans des chaînes. Ces méthodes retournent l'emplacement du texte recherché. Si le texte n'est pas trouvé, elles retournent `-1`. L'exemple suivant montre une recherche de la première et de la dernière occurrence du mot « methods » et affiche le texte qui se trouve entre les

deux.

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");
```

Recherche de texte spécifique en utilisant des expressions régulières

La classe System.Text.RegularExpressions.Regex permet d'effectuer des recherches de chaînes. Ces recherches peuvent varier en complexité et aller de modèles simples à des modèles plus compliqués.

L'exemple de code suivant recherche le mot « the » ou « their » dans une phrase, indépendamment de la casse. La méthode statique Regex.IsMatch effectue la recherche. Vous lui passez la chaîne à rechercher et un modèle de recherche. Dans ce cas, un troisième argument spécifie que la recherche est insensible à la casse. Pour plus d'informations, consultez System.Text.RegularExpressions.RegexOptions.

Le modèle de recherche décrit le texte que vous recherchez. Le tableau suivant décrit chaque élément du modèle de recherche. (Le tableau ci-dessous utilise le seul `\` qui doit être placé `\\` dans une séquence d'échappement comme dans une chaîne C#).

MODÈLE	SIGNIFICATION
`the`	correspond au texte « the »
`(eir)?`	correspond à 0 ou 1 occurrence de « eir »
`\s`	correspond à un espace blanc

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

TIP

Les méthodes `string` sont généralement de meilleurs choix quand vous recherchez une chaîne précise. Les expressions régulières sont mieux adaptées lorsque vous recherchez un modèle dans une chaîne source.

Une chaîne suit-elle un modèle ?

Le code suivant utilise des expressions régulières pour valider le format de chaque chaîne d'un tableau. La validation doit vérifier que chaque chaîne a la forme d'un numéro de téléphone constitué de trois groupes de chiffres séparés par des tirets, les deux premiers groupes contenant trois chiffres et le troisième en contenant quatre. Le modèle de recherche utilise l'expression régulière `^\\d{3}-\\d{3}-\\d{4}$`. Pour plus d'informations, consultez Langage des expressions régulières - Aide-mémoire.

MODÈLE	SIGNIFICATION
`^`	correspond au début de la chaîne
`\d{3}`	correspond à exactement 3 caractères numériques
`-`	correspond au caractère « - ».
`\d{3}`	correspond à exactement 3 caractères numériques
`-`	correspond au caractère « - ».
`\d{4}`	correspond à exactement 4 caractères numériques
`$`	correspond à la fin de la chaîne

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Ce seul modèle de recherche correspond à de nombreuses chaînes valides. Les expressions régulières sont mieux adaptées pour rechercher un modèle ou pour effectuer une validation par rapport à un modèle, et le sont moins pour une seule chaîne de texte.

Voir aussi

- Guide de programmation C#
- Chaînes
- LINQ et chaînes
- System.Text.RegularExpressions.Regex
- Expressions régulières .NET
- Langage des expressions régulières-aide-mémoire
- Bonnes pratiques pour l'utilisation de chaînes dans .NET

Comment modifier le contenu d'une chaîne en C#

02/11/2020 • 9 minutes to read • Edit Online

Cet article présente plusieurs techniques pour produire un `string` en modifiant un `string` existant. Toutes les techniques présentées retournent le résultat des modifications sous la forme d'un nouvel objet `string`. Pour démontrer que les chaînes d'origine et modifiées sont des instances distinctes, les exemples stockent le résultat dans une nouvelle variable. Vous pouvez examiner les originaux `string` et les nouveaux, modifiés `string` lorsque vous exécutez chaque exemple.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Plusieurs techniques sont présentées dans cet article. Vous pouvez remplacer du texte existant. Vous pouvez rechercher des modèles et remplacer le texte correspondant par un autre texte. Vous pouvez traiter une chaîne en tant que séquence de caractères. Vous pouvez également utiliser des méthodes pratiques qui suppriment les espaces blancs. Choisissez les techniques qui correspondent le mieux à votre scénario.

Remplacer du texte

Le code suivant crée une chaîne en remplaçant le texte existant par un substitut.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

Le code précédent illustre la propriété *immuable* des chaînes. Vous pouvez voir dans l'exemple précédent que la chaîne d'origine, `source`, n'est pas modifiée. La méthode String.Replace crée un `string` contenant les modifications.

La méthode Replace peut remplacer des chaînes ou des caractères uniques. Dans les deux cas, toutes les occurrences du texte recherché sont remplacées. L'exemple suivant remplace tous les caractères ' ' par '_' :

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

La chaîne source ne change pas, et une nouvelle chaîne est retournée avec le remplacement.

Supprimer les espaces blancs

Vous pouvez utiliser les méthodes String.Trim, String.TrimStart et String.TrimEnd pour supprimer les espaces blancs de début ou de fin. Le code suivant montre un exemple de chaque. La chaîne source ne change pas. Ces méthodes retournent une nouvelle chaîne avec le contenu modifié.

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Supprimer du texte

Vous pouvez supprimer du texte d'une chaîne à l'aide de la méthode String.Remove. Cette méthode supprime un certain nombre de caractères en commençant à un index spécifique. L'exemple suivant montre comment utiliser String.IndexOf suivi de Remove pour supprimer le texte d'une chaîne :

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Remplacer du texte correspondant à un modèle

Vous pouvez utiliser des expressions régulières pour remplacer du texte correspondant à un modèle par un autre texte, éventuellement défini par un modèle. L'exemple suivant utilise la classe System.Text.RegularExpressions.Regex pour trouver un modèle dans une chaîne source et le remplacer par du texte dont la casse est correcte. La méthode Regex.Replace(String, String, MatchEvaluator, RegexOptions) accepte dans ses arguments une fonction qui fournit la logique du remplacement. Dans cet exemple, la fonction `LocalReplaceMatchCase` est une **fonction locale** déclarée à l'intérieur de l'exemple de méthode. `LocalReplaceMatchCase` utilise la classe System.Text.StringBuilder pour générer la chaîne de remplacement avec la casse correcte.

Les expressions régulières sont plus utiles pour rechercher et remplacer du texte qui suit un modèle que du texte connu. Pour plus d'informations, consultez Comment rechercher des chaînes. Le modèle de recherche, « the\s », recherche le mot « the » suivi d'un espace blanc. Cette partie du modèle permet d'exclure le mot « there » qui figure dans la chaîne source. Pour plus d'informations sur les éléments du langage des expressions régulières, consultez Langage des expressions régulières - Aide-mémoire.

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

La méthode StringBuilder.ToString retourne une chaîne immuable avec le contenu dans l'objet StringBuilder.

Modification de caractères individuels

Vous pouvez générer un tableau de caractères à partir d'une chaîne, modifier le contenu du tableau, puis créer une autre chaîne à partir du contenu modifié du tableau.

L'exemple suivant montre comment remplacer un jeu de caractères dans une chaîne. Tout d'abord, il utilise la méthode String.ToCharArray() pour créer un tableau de caractères. Il utilise la méthode IndexOf pour trouver l'index de départ du mot « fox ». Les trois caractères suivants sont remplacés par un autre mot. Enfin, une nouvelle chaîne est construite à partir du tableau de caractères mis à jour.

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Créer un contenu de chaîne par programmation

Étant donné que les chaînes sont immuables, les exemples précédents créent tous des chaînes temporaires ou des tableaux de caractères. Dans les scénarios à hautes performances, il peut être souhaitable d'éviter ces allocations de tas. .NET Core fournit une String.Create méthode qui vous permet de remplir par programmation le contenu de caractères d'une chaîne via un rappel tout en évitant les allocations de chaînes temporaires intermédiaires.

```
// constructing a string from a char array, prefix it with some additional characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[] charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Vous pouvez modifier une chaîne dans un bloc fixed avec du code unsafe, mais il est **fortement** déconseillé de modifier le contenu de la chaîne après la création d'une chaîne. Cela entraînera une rupture des choses de manière imprévisible. Par exemple, si quelqu'un effectue une chaîne qui a le même contenu que le vôtre, il obtient votre copie et ne s'attend pas à ce que vous modifiiez sa chaîne.

Voir aussi

- Expressions régulières .NET
- Langage des expressions régulières-aide-mémoire

Comment comparer des chaînes en C#

02/11/2020 • 18 minutes to read • Edit Online

Vous comparez des chaînes pour répondre à une des deux questions : « Ces chaînes sont-elle égales ? » ou « Dans quel ordre ces chaînes doivent-elles être placées pendant le tri ? »

Ces deux questions sont compliquées en raison des facteurs qui affectent les comparaisons de chaînes :

- Vous pouvez choisir une comparaison ordinale ou linguistique.
- Vous pouvez choisir si la casse a une incidence.
- Vous pouvez choisir des comparaisons propres à la culture.
- Les comparaisons linguistiques dépendent de la culture et de la plateforme.

> **NOTE**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Quand vous comparez des chaînes, vous les ordonnez les unes par rapport aux autres. Les comparaisons permettent de trier une séquence de chaînes. Quand la séquence est dans un ordre connu, elle est plus facile à observer, que ce soit pour les logiciels et pour les humains. Les autres comparaisons peuvent vérifier si les chaînes sont identiques. Ces vérifications de similitude se rapprochent de celles d'égalité, sauf que certaines différences, comme celles qui concernent la casse, peuvent être ignorées.

Comparaisons ordinales par défaut

Par défaut, les opérations les plus courantes :

- String.CompareTo
- String.Equals
- String.Equalityet String.Inequality , autrement dit, les opérateurs d'égalité `==` et `!=` , respectivement

effectue une comparaison ordinale respectant la casse et, si nécessaire, utilise la culture actuelle. L'exemple suivant illustre ce qui suit :

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal")}");
```

La comparaison ordinale par défaut ne prend pas en compte les règles linguistiques lors de la comparaison de chaînes. Elle compare la valeur binaire de chaque objet Char dans les deux chaînes. La comparaison ordinale par défaut respecte ainsi également la casse.

Le test d'égalité avec String.Equals et les `==` `!=` opérateurs et diffère de la comparaison de chaînes à l'aide des String.CompareTo Compare(String, String) méthodes et. Tandis que les tests d'égalité effectuent une comparaison ordinale respectant la casse, les méthodes de comparaison effectuent une comparaison respectant la casse et dépendante de la culture à l'aide de la culture actuelle. Étant donné que les méthodes de comparaison par défaut effectuent souvent différents types de comparaisons, nous vous recommandons de toujours indiquer clairement l'intention de votre code en appelant une surcharge qui spécifie explicitement le type de comparaison à effectuer.

Comparaisons ordinales ne respectant pas la casse

La méthode String.Equals(String, StringComparison) vous permet d'indiquer une valeur StringComparison de StringComparison.OrdinalIgnoreCase pour une comparaison ordinale ne respectant pas la casse. Il existe également une méthode String.Compare(String, String, StringComparison) statique qui effectue une comparaison ordinale ne respectant pas la casse si vous spécifiez la valeur StringComparison.OrdinalIgnoreCase pour l'argument StringComparison. Elles sont représentées dans le code suivant :

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType: StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not
equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

Lorsque vous effectuez une comparaison ordinale ne respectant pas la casse, ces méthodes utilisent les conventions de casse de la culture invariante.

Comparaisons linguistiques

Les chaînes peuvent aussi être triées à l'aide de règles linguistiques pour la culture actuelle. C'est ce qu'on appelle parfois « Ordre de tri par mot ». Quand vous effectuez une comparaison linguistique, certains caractères Unicode non alphanumériques peuvent avoir une pondération spéciale. Par exemple, le trait d'Union « - » peut avoir un poids faible affecté, de sorte que « co-op » et « Coop » s'affichent à côté l'un de l'autre dans l'ordre de tri. Par ailleurs, certains caractères Unicode peuvent être équivalents à une séquence d'instances Char. L'exemple suivant utilise la phrase « Ils dansent dans la rue » en allemand avec « ss » (U+0073 U+0073) dans une chaîne et « ß » (U+00DF) dans une autre. Du point de vue linguistique (dans Windows), « ss » équivaut à l'Esszet allemand : le caractère « ß » dans les cultures « en-US » et « de-DE ».

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Cet exemple illustre les comparaisons linguistiques dépendantes du système d'exploitation. L'hôte de la fenêtre interactive est un hôte Linux. Les comparaisons linguistiques et ordinales produisent les mêmes résultats. Si vous exécutez ce même exemple sur un hôte Windows, la sortie suivante s'affiche :

```
<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison
```

Sur Windows, l'ordre de tri pour « cop », « coop » et « co-op » change quand vous passez d'une comparaison linguistique à une comparaison ordinale. De même, les deux phrases en allemand ne sont pas comparées de la même façon selon le type de comparaison.

Comparaisons utilisant des cultures spécifiques

Cet exemple stocke des objets CultureInfo pour les cultures en-US et de-DE. Les comparaisons sont effectuées à l'aide de l'un objet CultureInfo pour garantir une comparaison propre à la culture.

La culture utilisée affecte les comparaisons linguistiques. L'exemple suivant montre les résultats de la comparaison des deux phrases en allemand avec la culture « en-US » et la culture « de-DE » :

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Les comparaisons dépendantes de la culture sont généralement utilisées pour comparer et trier des chaînes entrées par plusieurs utilisateurs. Les caractères et les conventions de tri de ces chaînes peuvent varier selon les paramètres régionaux de l'ordinateur de l'utilisateur. Même les chaînes qui contiennent des caractères identiques peuvent être triées différemment selon la culture du thread actuel. En outre, essayez cet exemple de code localement sur un ordinateur Windows et vous obtiendrez les résultats suivants :

```
<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison
```

Les comparaisons linguistiques sont dépendantes de la culture actuelle et du système d'exploitation. Prenez cela en compte lorsque vous travaillez avec des comparaisons de chaînes.

Tri linguistique et recherche de chaînes dans des tableaux

Les exemples suivants montrent comment trier et rechercher des chaînes dans un tableau à l'aide d'une comparaison linguistique dépendante de la culture actuelle. Vous utilisez les méthodes statiques Array qui prennent un paramètre System.StringComparer.

Cet exemple montre comment trier un tableau de chaînes à l'aide de la culture actuelle :

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Une fois que le tableau est trié, vous pouvez rechercher des entrées à l'aide d'une recherche binaire. Une recherche binaire commence au milieu de la collection pour déterminer la moitié de la collection qui doit contenir la chaîne recherchée. Chaque comparaison ultérieure divise à son tour en deux la partie restante de la collection. Le tableau est trié à l'aide de StringComparer.CurrentCulture. La fonction locale `ShowWhere` affiche des informations sur l'emplacement de la chaîne. Si la chaîne est introuvable, la valeur retournée indique où elle serait si elle a été trouvée.

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Tri ordinal et recherche dans des collections

Le code suivant utilise la classe de collection System.Collections.Generic.List<T> pour stocker les chaînes. Les chaînes sont triées à l'aide de la méthode List<T>.Sort. Cette méthode a besoin d'un délégué qui compare et trie les deux chaînes. La méthode String.CompareTo fournit cette fonction de comparaison. Exécutez l'exemple et observez l'ordre. Cette opération de tri utilise un tri ordinal qui respecte la casse. Vous devez utiliser les méthodes statiques String.Compare pour spécifier des règles de comparaison différentes.

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Une fois triée, la liste de chaînes peut être parcourue à l'aide d'une recherche binaire. L'exemple suivant montre comment parcourir la liste triée à l'aide de la même fonction de comparaison. La fonction locale `ShowWhere` montre où se trouve le texte recherché ou là où il devrait se trouver :

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Vérifiez toujours que vous utilisez le même type de comparaison pour le tri et la recherche. L'utilisation de différents types de comparaison pour le tri et la recherche produit des résultats inattendus.

Les classes de collection telles que System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue> et System.Collections.Generic.List<T> ont des constructeurs qui prennent un paramètre System.StringComparer quand le type des éléments ou des clés est `string`. En général, vous devez utiliser ces constructeurs chaque fois que cela vous est possible, et spécifier StringComparer.Ordinal ou StringComparer.OrdinalIgnoreCase.

Égalité de référence et centralisation des chaînes

Aucun des exemples n'a utilisé ReferenceEquals. Cette méthode détermine si deux chaînes sont le même objet, ce qui peut entraîner des résultats incohérents dans les comparaisons de chaînes. L'exemple suivant illustre la fonctionnalité de *centralisation des chaînes* du langage C#. Lorsqu'un programme déclare plusieurs variables de chaînes identiques, le compilateur les stocke au même emplacement. En appelant la méthode ReferenceEquals, vous pouvez voir que les deux chaînes référencent le même objet en mémoire. Utilisez la méthode String.Copy pour éviter la centralisation. Une fois la copie effectuée, les deux chaînes ont différents emplacements de stockage, même si elles ont la même valeur. Exécutez l'exemple suivant pour montrer que les chaînes `a` et `b` sont *centralisées*, ce qui signifie qu'elles partagent le même stockage. Les chaînes `a` et `c` ne le sont pas.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

Quand vous recherchez l'égalité des chaînes, vous devez utiliser les méthodes qui spécifient explicitement le type de comparaison que vous souhaitez effectuer. Votre code est plus lisible et plus facile à gérer. Utilisez les surcharges des méthodes des classes System.String et System.Array qui prennent un paramètre d'énumération StringComparison. Vous spécifiez le type de comparaison à effectuer. Évitez d'utiliser les opérateurs `==` et `!=` dans la recherche d'égalité. Les méthodes d'instance String.CompareTo effectuent toujours une comparaison ordinale respectant la casse. Elles sont principalement adaptées au classement des chaînes par ordre alphabétique.

Vous pouvez intégrer une chaîne ou récupérer une référence à une chaîne intégrée existante en appelant la méthode String.Intern. Pour déterminer si une chaîne est intégrée, appelez la méthode String.IsInterned.

Voir aussi

- System.Globalization.CultureInfo
- System.StringComparer
- Chaînes
- Comparaison de chaînes
- Globalisation et localisation d'applications

Comment effectuer un cast en toute sécurité à l'aide de critères spéciaux et des opérateurs is et As

02/11/2020 • 7 minutes to read • Edit Online

Dans la mesure où les objets sont polymorphes, une variable d'un type de classe de base peut contenir un type dérivé. Pour accéder aux membres d'instance du type dérivé, il est nécessaire de réeffectuer un cast de la valeur vers le type dérivé. Toutefois, un cast risque de lever InvalidCastException. C# fournit des instructions relatives aux critères spéciaux, qui effectuent un cast de manière conditionnelle, uniquement en cas de réussite. C# fournit également les opérateurs is et as pour tester l'appartenance d'une valeur à un certain type.

L'exemple suivant montre comment utiliser l'instruction de critères spéciaux `is` :

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        var g = new Giraffe();
        var a = new Animal();
        FeedMammals(g);
        FeedMammals(a);
        // Output:
        // Eating.
        // Animal is not a Mammal

        SuperNova sn = new SuperNova();
        TestForMammals(g);
        TestForMammals(sn);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    static void FeedMammals(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
        else
        {
            // variable 'm' is not in scope here, and can't be used.
            Console.WriteLine($"{a.GetType().Name} is not a Mammal");
        }
    }

    static void TestForMammals(object o)
    {
        // You also can use the as operator and test for null
        // before referencing the variable.
        var m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }
}
```

L'exemple précédent illustre quelques fonctionnalités de la syntaxe relative aux critères spéciaux. L' `if (a is Mammal m)` instruction associe le test à une assignation d'initialisation. L'assignation n'a lieu qu'en cas de réussite du test. La variable `m` se trouve uniquement dans l'étendue de l'instruction `if` incorporée, à laquelle elle

a été assignée. Vous ne pouvez pas accéder à `m` plus tard dans la même méthode. L'exemple précédent montre également comment utiliser l' `as` opérateur pour convertir un objet en un type spécifié.

Vous pouvez également utiliser la même syntaxe pour le test si un type valeur Nullable a une valeur, comme illustré dans l'exemple suivant :

```
class Program
{
    static void Main(string[] args)
    {
        int i = 5;
        PatternMatchingNullable(i);

        int? j = null;
        PatternMatchingNullable(j);

        double d = 9.78654;
        PatternMatchingNullable(d);

        PatternMatchingSwitch(i);
        PatternMatchingSwitch(j);
        PatternMatchingSwitch(d);
    }

    static void PatternMatchingNullable(System.ValueType val)
    {
        if (val is int j) // Nullable types are not allowed in patterns
        {
            Console.WriteLine(j);
        }
        else if (val is null) // If val is a nullable type with no value, this expression is true
        {
            Console.WriteLine("val is a nullable type with the null value");
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }

    static void PatternMatchingSwitch(System.ValueType val)
    {
        switch (val)
        {
            case int number:
                Console.WriteLine(number);
                break;
            case long number:
                Console.WriteLine(number);
                break;
            case decimal number:
                Console.WriteLine(number);
                break;
            case float number:
                Console.WriteLine(number);
                break;
            case double number:
                Console.WriteLine(number);
                break;
            case null:
                Console.WriteLine("val is a nullable type with the null value");
                break;
            default:
                Console.WriteLine("Could not convert " + val.ToString());
                break;
        }
    }
}
```

L'exemple précédent illustre d'autres fonctionnalités relatives aux critères spéciaux pour les conversions. Vous pouvez tester le modèle null d'une variable en vérifiant spécifiquement la valeur `null`. Quand la valeur runtime de la variable est `null`, une instruction `is` de vérification de type retourne toujours `false`. L'instruction `is`

relative aux critères spéciaux n'autorise aucun type valeur Nullable, par exemple `int?` ou `Nullable<int>`. Toutefois, vous pouvez tester un autre type valeur. Les `is` modèles de l'exemple précédent ne sont pas limités aux types valeur Nullable. Vous pouvez également utiliser ces modèles pour tester si une variable d'un type référence a une valeur ou s'il s'agit de `null` .

L'exemple précédent montre également comment utiliser le modèle de type dans une `switch` instruction où la variable peut être l'un des nombreux types différents.

Si vous souhaitez tester si une variable est un type donné, mais ne pas l'assigner à une nouvelle variable, vous pouvez utiliser les `is` opérateurs et pour les types `as` référence et les types valeur Nullable. Le code suivant montre comment utiliser les instructions `is` et `as`, qui faisaient partie du langage C# avant l'introduction des critères spéciaux permettant de déterminer l'existence ou non d'un type donné pour une variable :

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        UseAsOperator(sn);

        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        UseAsWithNullable(i);

        double d = 9.78654;
        UseAsWithNullable(d);
    }

    static void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    static void UsePatternMatchingIs(Animal a)
    {
```

```
        if (a is Mammal m)
        {
            m.Eat();
        }
    }

    static void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }

    static void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }
}
```

Comme vous pouvez le constater en comparant ce code avec le code utilisant les critères spéciaux, la syntaxe des critères spéciaux fournit des fonctionnalités plus robustes, car elle combine le test et l'assignation dans une seule instruction. Utilisez la syntaxe des critères spéciaux autant que possible.

Kit SDK .NET Compiler Platform

02/11/2020 • 12 minutes to read • Edit Online

Les compilateurs génèrent un modèle détaillé du code applicatif lorsqu'ils en valident la syntaxe et la sémantique. Ce modèle permet de générer la sortie exécutable à partir du code source. Le Kit SDK .NET Compiler Platform donne accès à ce modèle. Nous nous appuyons de plus en plus sur des fonctionnalités d'environnements de développement intégré (IDE) telles qu'IntelliSense, la refactorisation, le renommage intelligent, la recherche de l'ensemble des références et l'accès direct aux définitions pour augmenter notre productivité. Nous utilisons des outils d'analyse du code pour améliorer la qualité de notre code et des générateurs de code pour faciliter la construction des applications. Plus ces outils gagnent en intelligence, plus ils ont besoin d'accéder à une grande part du modèle que seuls les compilateurs créent lorsqu'ils traitent le code applicatif. Il s'agit de la mission principale des API Roslyn : l'ouverture des zones opaques et l'autorisation des outils et des utilisateurs finaux à partager au sein de la multitude de compilateurs d'informations sur notre code. Au lieu de rester d'opaques traducteurs de code source en code objet, les compilateurs deviennent des plateformes avec Roslyn : des API utilisables pour les tâches portant sur le code dans les outils et les applications souhaités.

Concepts du Kit SDK .NET Compiler Platform

Le Kit SDK .NET Compiler Platform réduit considérablement la barrière à l'entrée pour la création d'applications et d'outils axés sur le code. Il crée de nombreuses opportunités d'innovation dans des domaines tels que la méta-programmation, la génération et la transformation de code, l'utilisation interactive des langages C# et Visual Basic, ainsi que l'incorporation de C# et de Visual Basic dans des langages spécifiques à un domaine.

Le Kit SDK .NET Compiler Platform permet de générer des ***analyseurs*** et des ***correctifs de code*** qui recherchent et corrigent les erreurs de codage. Les ***analyseurs*** comprennent la syntaxe et la structure du code, et détectent les pratiques à corriger. Les ***correctifs de code*** représentent des propositions de correction des erreurs de codage trouvées par les analyseurs. En règle générale, un analyseur et les correctifs de code associés sont regroupés dans un seul projet.

Les analyseurs et les correctifs de code utilisent l'analyse statique pour comprendre le code. Ils n'exécutent pas le code, et n'offrent aucun autre avantage en matière de tests. Toutefois, ils peuvent signaler des pratiques qui mènent souvent à des bogues, à un code non gérable ou à une violation des indications standard.

Le Kit SDK .NET Compiler Platform se compose d'un unique ensemble d'API permettant d'examiner et de comprendre un codebase C# ou Visual Basic. Grâce à ce codebase unique, il est plus facile d'écrire des analyseurs et des correctifs de code en utilisant les API d'analyse syntaxique et sémantique fournies par le Kit SDK .NET Compiler Platform. Une fois libéré de la tâche chronophage qui consiste à répliquer l'analyse effectuée par le compilateur, vous pouvez vous concentrer sur la tâche, plus ciblée, de recherche et de résolution des erreurs de codage courantes de votre projet ou de votre bibliothèque.

Autre avantage, plus modeste, vos analyseurs et vos correctifs de code sont plus petits et utilisent beaucoup moins de mémoire une fois chargés dans Visual Studio que si vous aviez écrit votre propre codebase pour comprendre le code d'un projet. En exploitant les mêmes classes que le compilateur et Visual Studio, vous pouvez créer vos propres outils d'analyse statique. Votre équipe a donc la possibilité d'utiliser des analyseurs et des correctifs de code sans impact perceptible sur les performances de l'IDE.

Il existe trois grands scénarios d'écriture d'analyseurs et de correctifs de code :

1. *Appliquer des normes de codage pour l'équipe*
2. *Offrir de l'aide sur les packages de bibliothèque*
3. *Proposer des conseils généraux*

Appliquer des normes de codage pour l'équipe

De nombreuses équipes ont des normes de codage, qui sont appliquées par le biais des révisions du code effectuées avec d'autres membres de l'équipe. Les analyseurs et les correctifs de code peuvent rendre ce processus beaucoup plus efficace. Les revues de code se produisent quand un développeur partage son travail avec d'autres membres de l'équipe. Le développeur aura investi tout le temps nécessaire pour terminer une nouvelle fonctionnalité avant de recevoir le moindre commentaire. Plusieurs semaines peuvent s'écouler, au cours desquelles le développeur renforce des habitudes qui ne correspondent pas aux pratiques de l'équipe.

Les analyseurs s'exécutent au moment même où il écrit du code. Le développeur obtient immédiatement des commentaires qui l'encouragent à suivre les instructions sans attendre. Le développeur prend l'habitude d'écrire du code conforme dès qu'il commence le prototypage. Lorsque la fonctionnalité est prête à être révisée par des humains, toutes les instructions standard ont été appliquées.

Les équipes peuvent créer des analyseurs et des correctifs de code permettant de rechercher les pratiques les plus courantes qui enfreignent les pratiques de codage de l'équipe. Ils peuvent être installés sur l'ordinateur de chaque développeur pour appliquer les normes.

Offrir de l'aide sur les packages de bibliothèque

De nombreuses bibliothèques sont disponibles pour les développeurs .NET sur NuGet. Certaines proviennent de Microsoft ou de sociétés tierces, d'autres sont proposées par des membres de la communauté et des bénévoles. Ces bibliothèques sont plus souvent utilisées et sont mieux notées lorsque les développeurs réussissent à les exploiter.

En plus de la documentation, vous pouvez proposer des analyseurs et des correctifs de code qui recherchent et corrigent les mauvais emplois courants de votre bibliothèque. Ces corrections immédiates aident les développeurs à en tirer parti plus rapidement.

Vous pouvez ajouter les analyseurs et les correctifs de code au package de votre bibliothèque sur NuGet. Dans ce scénario, un développeur qui installe votre package NuGet installera également le package de l'analyseur. Tous les développeurs qui utilisent votre bibliothèque obtiennent immédiatement de l'aide de la part de votre équipe sous la forme de commentaires immédiats sur les erreurs et de suggestions de corrections.

Proposer des conseils généraux

La communauté de développeurs .NET a découvert, par l'expérience, des modèles qui fonctionnent bien et d'autres qu'il vaut mieux éviter. Plusieurs membres de la communauté ont créé des analyseurs qui appliquent ces modèles recommandés. Plus nous gagnons en expérience, plus il y a de place pour de nouvelles idées.

Ces analyseurs peuvent être chargés sur Visual Studio Marketplace et téléchargés par les développeurs avec Visual Studio. Ceux qui débutent avec le langage et la plateforme apprennent rapidement les pratiques acceptées et deviennent plus vite productifs, dès le début de leur parcours d'apprentissage de .NET. Comme ils sont de plus en plus utilisés, la communauté adopte ces pratiques.

Étapes suivantes

Le Kit SDK .NET Compiler Platform inclut les derniers modèles objet du langage pour la génération, l'analyse et la refactorisation de code. Cette section offre une vue d'ensemble conceptuelle du Kit SDK .NET Compiler Platform. Vous trouverez plus de détails dans les sections Démarrages rapides, exemples et didacticiels.

Pour en savoir plus sur les concepts du Kit SDK .NET Compiler Platform, consultez ces cinq rubriques :

- Explorer le code avec le visualiseur de syntaxe
- Comprendre le modèle de l'API du compilateur

- Utiliser la syntaxe
- Utiliser la sémantique
- Utiliser un espace de travail

Pour commencer, vous devez installer le kit **.NET Compiler Platform SDK** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Comprendre le modèle du SDK .NET Compiler Platform

02/11/2020 • 7 minutes to read • Edit Online

Les compilateurs traitent le code que vous écrivez en suivant des règles structurées qui diffèrent souvent de la façon dont vous pouvez vous-même lire et comprendre le code. Pour bien comprendre le fonctionnement des API dont vous avez besoin pour créer des outils Roslyn, vous devez connaître les concepts de base du modèle utilisé par les compilateurs.

Zones fonctionnelles du pipeline de compilateur

Le SDK .NET Compiler Platform expose les analyses de code des compilateurs C# et Visual Basic en vous fournissant, en tant que consommateur, une couche d'API qui reflète un pipeline de compilateur standard.

Chaque phase de ce pipeline correspond à un composant distinct. Tout d'abord, la phase d'analyse segmente et analyse le texte source dans la syntaxe grammaticale du langage utilisé. Ensuite, la phase de déclaration analyse les métadonnées sources et importées pour former des symboles nommés. Après cela, la phase de liaison mappe les identificateurs dans le code à des symboles. Enfin, la phase d'émission émet un assembly qui contient toutes les informations générées par le compilateur.

Pour chacune de ces phases, le SDK .NET Compiler Platform expose un modèle objet qui permet d'accéder aux informations disponibles à la phase en question. La phase d'analyse expose une arborescence de syntaxe, la phase de déclaration expose une table de symboles hiérarchique, la phase de liaison expose le résultat de l'analyse sémantique du compilateur, et la phase d'émission est une API qui produit des codes d'octet de langage intermédiaire.

Chaque compilateur combine ces composants en une solution de bout en bout unique.

Ces API sont les mêmes que celles utilisées par Visual Studio. Par exemple, les fonctionnalités de mise en forme et de mise en forme du code utilisent les arborescences de syntaxe, l' **Explorateur d'objets**et les fonctionnalités de navigation utilisent la table de symboles, les refactorisations et **atteindre la définition** utilisent le modèle sémantique, et **Modifier & Continuer** utilise tous ces éléments, y compris l'API d'émission.

Couches d'API

Le SDK du compilateur .NET est constitué de plusieurs couches d'API : les API du compilateur, les API de diagnostic, les API de script et les API d'espaces de travail.

API du compilateur

La couche du compilateur contient les modèles objet qui correspondent aux informations syntaxiques et sémantiques exposées à chaque phase du pipeline du compilateur. La couche du compilateur contient également un instantané immuable d'un appel unique d'un compilateur, notamment les références d'assembly, les options du compilateur et les fichiers de code source. Le langage C# et le langage Visual Basic sont représentés par deux API distinctes. Les deux API sont similaires, mais elles sont adaptées à une haute fidélité pour chaque langue. Cette couche n'a pas de dépendances sur les composants Visual Studio.

API de diagnostic

Dans le cadre de son analyse, le compilateur peut produire un ensemble de diagnostics complets, qui couvrent les erreurs de syntaxe, de sémantique et d'assignation définie, ainsi que divers avertissements et informations. La couche API du compilateur expose les diagnostics par le biais d'une API extensible qui permet d'intégrer des analyseurs définis par l'utilisateur au processus de compilation. Des diagnostics définis par l'utilisateur, tels que ceux générés par des outils comme StyleCop ou FxCop, peuvent ainsi être produits en même temps que les diagnostics définis par le compilateur. La production de diagnostics de cette façon présente l'avantage d'intégrer naturellement des outils tels que MSBuild et Visual Studio, qui dépendent des diagnostics pour des expériences telles que l'arrêt d'une build basée sur la stratégie et l'affichage des tildes dynamiques dans l'éditeur et la suggestion de correction de code.

API de script

Les API d'hébergement et de script font partie de la couche du compilateur. Vous pouvez utiliser ces API pour l'exécution d'extraits de code et l'accumulation d'un contexte d'exécution du runtime. Le REPL (Read-Evaluate-Print Loop) interactif C# utilise ces API. Le REPL vous permet d'utiliser C# comme langage de script, en exécutant le code que vous écrivez de manière interactive.

API des espaces de travail

La couche des espaces de travail contient les API des espaces de travail, à partir desquelles sont effectuées les analyses de code et la refactorisation dans les solutions entières. Il vous aide à organiser toutes les informations sur les projets d'une solution en un modèle d'objet unique, vous offrant un accès direct aux modèles d'objet de la couche du compilateur sans avoir à analyser des fichiers, configurer des options ou gérer des dépendances entre projets.

De plus, la couche des espaces de travail expose un ensemble d'API utilisées pour l'implémentation d'outils d'analyse du code et de refactorisation qui fonctionnent dans un environnement hôte, tel que l'IDE Visual Studio. Les API Rechercher toutes les références, Mise en forme et Génération de code en sont des exemples.

Cette couche n'a pas de dépendances sur les composants Visual Studio.

Utiliser la syntaxe

13/05/2020 • 17 minutes to read • Edit Online

Une *arborescence de syntaxe* est une structure de données de base qui est exposée par les API du compilateur. Ces arborescences représentent la structure lexicale et syntaxique du code source. Elles ont deux utilités principales :

- Pour autoriser les outils, tels qu'un IDE, des compléments, des outils d'analyse du code et des refactorisations, pour afficher et traiter la structure syntaxique du code source dans le projet d'un utilisateur.
- Pour activer les outils, tels que les refactorisations et un IDE, pour créer, modifier et réorganiser le code source de manière naturelle sans avoir à utiliser des modifications de texte direct. Par la création et la manipulation des arborescences, les outils peuvent facilement créer et réorganiser le code source.

Arborescences de syntaxe

Les arborescences de syntaxe sont la structure principale utilisée pour la compilation, l'analyse de code, la liaison, la refactorisation, les fonctionnalités de l'IDE et la génération de code. Aucune partie du code source ne peut être comprise sans avoir au préalable été identifiée et classée dans l'un des nombreux éléments de langage structurel connus.

Les arborescences de syntaxe ont trois caractéristiques principales. Leur première caractéristique est qu'elles stockent toutes les informations sources avec une haute fidélité. Une fidélité complète signifie que l'arborescence de syntaxe contient chaque information trouvée dans le texte source, chaque construction grammaticale, chaque jeton lexical et tout le reste entre, y compris les espaces blancs, les commentaires et les directives de préprocesseur. Par exemple, chaque littéral mentionné dans la source est représenté exactement comme il a été tapé. Les arborescences de syntaxe capturent également les erreurs dans le code source lorsque le programme est incomplet ou incorrect en représentant des jetons ignorés ou manquants.

Le deuxième attribut des arborescences de syntaxe est qu'ils peuvent produire le texte exact à partir duquel ils ont été analysés. À partir de n'importe quel nœud de syntaxe, il est possible d'extraire la représentation textuelle de la sous-arborescence enracinée sur ce nœud. Cette capacité signifie que les arborescences de syntaxe peuvent être utilisées pour construire et modifier le texte source. En créant une arborescence, vous avez créé le texte équivalent et, en modifiant une arborescence de syntaxe, en rendant une nouvelle arborescence des modifications apportées à une arborescence existante, vous avez modifié le texte.

La troisième caractéristique des arborescences de syntaxe est qu'elles sont immuables et thread-safe. Après l'obtention d'une arborescence, il s'agit d'un instantané de l'état actuel du code et ne change jamais. Cela permet à plusieurs utilisateurs d'interagir simultanément avec la même arborescence de syntaxe dans différents threads sans verrouillage ou duplication. Étant donné que les arborescences sont immuables et qu'elles ne peuvent pas être modifiées directement, les méthodes de fabrique facilitent la création et la modification des arborescences de syntaxe en créant des instantanés supplémentaires de l'arborescence. Les arborescences réutilisent avantageusement les nœuds sous-jacents pour accélérer la génération d'une nouvelle version, sans consommer beaucoup plus de mémoire.

Une arborescence de syntaxe est littéralement une structure de données arborescente, où des éléments structurels non terminaux sont parents d'autres éléments. Chaque arborescence de syntaxe est constituée de nœuds, de jetons et de trivia.

Nœuds de syntaxe

Les nœuds de syntaxe figurent parmi les principaux éléments des arborescences de syntaxe. Ces nœuds représentent des constructions syntaxiques, telles que des déclarations, des instructions, des clauses et des

expressions. Chaque catégorie de nœuds de syntaxe est représentée par une classe distincte dérivée de Microsoft.CodeAnalysis.SyntaxNode. L'ensemble de classes de nœud n'est pas extensible.

Tous les nœuds de syntaxe sont des nœuds non terminaux dans l'arborescence de syntaxe ; ils ont donc toujours d'autres nœuds et jetons comme enfants. Chaque nœud enfant d'un autre nœud a un nœud parent qui est accessible via la propriété SyntaxNode.Parent. Comme les nœuds et les arborescences sont immuables, le parent d'un nœud ne change jamais. La racine de l'arborescence a un parent Null.

Chaque nœud a une méthode SyntaxNode.ChildNodes(), qui retourne une liste des nœuds enfants dans l'ordre séquentiel déterminé par leur position dans le texte source. Cette liste ne contient pas de jetons. Chaque nœud a également des méthodes permettant d'examiner les descendants, tels que DescendantNodes , DescendantTokens ou, DescendantTrivia qui représentent une liste de tous les nœuds, jetons ou anecdotes qui existent dans la sous-arborescence associée à ce nœud.

En outre, chaque sous-classe de nœud de syntaxe expose les mêmes enfants par le biais de propriétés fortement typées. Par exemple, une classe de nœud BinaryExpressionSyntax a trois propriétés supplémentaires propres aux opérateurs binaires : Left, OperatorToken et Right. Le type de Left et Right est ExpressionSyntax, et le type de OperatorToken est SyntaxToken.

Certains nœuds de syntaxe ont des enfants facultatifs. Par exemple, IfStatementSyntax a un ElseClauseSyntax facultatif. Si l'enfant n'est pas trouvé, la propriété retourne la valeur Null.

Jetons de syntaxe

Les jetons de syntaxe sont les éléments terminaux dans la grammaire d'un langage ; ils représentent les plus petits fragments syntaxiques du code. Ils ne sont jamais parents d'autres nœuds ou jetons. Les jetons de syntaxe se composent de mots clés, d'identificateurs, de littéraux et de caractères de ponctuation.

Par souci d'efficacité, le type SyntaxToken est un type valeur CLR. Par conséquent, contrairement aux nœuds de syntaxe, il n'y a qu'une seule structure pour tous les genres de jetons, avec une combinaison de propriétés dont le sens dépend du genre du jeton représenté.

Par exemple, un jeton de littéral d'entier représente une valeur numérique. En plus du texte source brut sur lequel s'étend le jeton, le jeton de littéral a une propriété Value qui indique précisément la valeur entière décodée. Cette propriété est typée en Object, car elle peut être l'un des nombreux types primitifs.

La propriété ValueText fournit les mêmes informations que la propriété Value, mais elle est toujours typée en String. Dans du texte source C#, un identificateur peut inclure des caractères d'échappement Unicode, mais la syntaxe de la séquence d'échappement n'est pas considérée comme faisant partie du nom de l'identificateur. Même si le texte brut sur lequel s'étend le jeton inclut la séquence d'échappement, la propriété ValueText ne l'inclut pas. À la place, elle inclut les caractères Unicode identifiés par la séquence d'échappement. Par exemple, si le texte source contient un identificateur écrit sous la forme `\u03C0`, la propriété ValueText pour ce jeton retourne `π`.

Trivia de syntaxe

Les trivia de syntaxe représentent les parties du texte source qui n'ont pas une grande utilité pour la compréhension générale du code. Il s'agit notamment des espaces blancs, des commentaires et des directives de préprocesseur. Comme les jetons de syntaxe, les trivia sont des types valeur. Le type Microsoft.CodeAnalysis.SyntaxTrivia unique est utilisé pour décrire tous les genres de trivia.

Du fait que les trivia ne font pas partie de la syntaxe du langage à proprement parler et qu'ils peuvent se trouver n'importe où entre deux jetons, ils ne sont pas représentés dans l'arborescence de syntaxe comme enfants d'un nœud. Les trivia sont toutefois inclus dans l'arborescence de syntaxe, car ils sont importants lors de l'implémentation d'une fonctionnalité telle que la refactorisation, mais aussi pour garantir une haute fidélité avec le texte source.

Vous pouvez accéder à la anecdote en inspectant le ou les collections d'un jeton SyntaxToken.LeadingTrivia SyntaxToken.TrailingTrivia . Quand le texte source est analysé, les séquences de trivia sont associées aux jetons. En règle générale, un jeton est propriétaire de tous les trivia qui se trouvent après lui sur la même ligne. Les trivia situés après cette ligne dépendent du jeton suivant. Dans le fichier source, le premier jeton obtient tous les trivia initiaux, et la dernière séquence de trivia est associée au jeton de fin de fichier, qui a sinon une largeur égale à zéro.

Contrairement aux nœuds et jetons de syntaxe, les trivia de syntaxe n'ont pas de parents. Toutefois, comme les trivia font partie de l'arborescence et que chacun d'eux est associé à un jeton unique, vous pouvez accéder au jeton correspondant à l'aide de la propriété SyntaxTrivia.Token.

Étendues

Chaque nœud, jeton ou trivia connaît sa position dans le texte source et le nombre de caractères qui le composent. La position du texte est représentée par un entier 32 bits, qui est un index `char` de base zéro. L'objet TextSpan représente la position de début et le nombre de caractères sous forme de deux entiers. Si TextSpan a une longueur nulle, il fait référence à une position entre deux caractères.

Chaque nœud a deux propriétés TextSpan : Span et FullSpan.

La Span propriété est l'étendue de texte à partir du début du premier jeton dans la sous-arborescence du nœud jusqu'à la fin du dernier jeton. Cette étendue n'inclut pas les trivia de début ou de fin.

La FullSpan propriété est l'étendue de texte qui comprend l'étendue normale du nœud, plus l'étendue des anecdotes de début ou de fin.

Par exemple :

```
        if (x > 3)
        {
||          // this is bad
            |throw new Exception("Not right.");|  // better exception?||
        }
```

L'étendue du nœud de l'instruction dans le bloc est délimitée par deux barres verticales simples (|). Elle inclut les caractères `throw new Exception("Not right.");`. L'étendue complète est délimitée par les deux barres verticales doubles (||). Elle inclut les mêmes caractères que l'étendue du nœud, plus les caractères associés aux trivia de début et de fin.

Genres

Chaque nœud, jeton ou trivia a une propriété SyntaxNode.RawKind de type System.Int32, qui identifie précisément l'élément de syntaxe représenté. Cette valeur peut être convertie en une énumération spécifique au langage. Chaque langage, C# ou Visual Basic, possède une `SyntaxKind` énumération unique (Microsoft.CodeAnalysis.CSharp.SyntaxKind et Microsoft.CodeAnalysis.VisualBasic.SyntaxKind , respectivement) qui répertorie tous les nœuds, jetons et éléments de anecdotes possibles dans la grammaire. Cette conversion peut être effectuée automatiquement en accédant aux méthodes d'extension CSharpExtensions.Kind ou VisualBasicExtensions.Kind.

La propriété RawKind permet de lever facilement toute ambiguïté sur les types de nœud de syntaxe qui utilisent la même classe de nœud. Pour les jetons et les trivia, cette propriété est le seul moyen de différencier les types d'élément entre eux.

Prenons l'exemple d'une classe BinaryExpressionSyntax, qui a les enfants Left, OperatorToken et Right. La propriété Kind détermine si le nœud de syntaxe est du genre AddExpression, SubtractExpression ou MultiplyExpression.

Errors

Même si le texte source contient des erreurs de syntaxe, une arborescence de syntaxe complète avec aller-retour au code source est exposée. Lorsque l'analyseur rencontre du code qui n'est pas conforme à la syntaxe définie du langage, il utilise l'une des deux techniques suivantes pour créer une arborescence de syntaxe :

- Si l'analyseur attend un type particulier de jeton, mais ne le trouve pas, il peut insérer un jeton manquant dans l'arborescence de syntaxe à l'emplacement où le jeton était attendu. Un jeton manquant représente le jeton qui était attendu, mais du fait qu'il a une étendue vide, sa propriété SyntaxNode.IsMissing retourne `true`.
- L'analyseur peut ignorer les jetons jusqu'à ce qu'il en trouve un dans lequel il peut continuer l'analyse. Dans ce cas, les jetons ignorés sont attachés en tant que nœud de trivia avec le genre SkippedTokensTrivia.

Utiliser la sémantique

18/03/2020 • 7 minutes to read • Edit Online

Les arborescences de syntaxe représentent la structure lexicale et syntaxique du code source. Cette information est suffisante pour décrire toutes les déclarations et la logique utilisées dans le code source, mais elle ne l'est pas pour identifier l'élément référencé. Un nom peut représenter les éléments suivants :

- Un type
- Un champ
- Une méthode
- Une variable locale

Ces éléments se distinguent par des caractéristiques uniques. Toutefois, pour déterminer à quel élément un identificateur fait réellement référence, avoir une connaissance approfondie des règles de langage s'avère souvent indispensable.

Certains éléments de programme sont représentés dans le code source, mais les programmes peuvent aussi faire référence à des bibliothèques précédemment compilées, empaquetées dans des fichiers d'assembly. Il n'y a pas de code source et donc pas d'arborescences ni nœuds de syntaxe disponibles pour les assemblys, mais les programmes peuvent faire référence à des éléments qu'ils contiennent.

Pour ces tâches, vous devez utiliser le **modèle sémantique**.

Par rapport à un modèle syntaxique du code source, un modèle sémantique encapsule en plus les règles de langage, ce qui vous aide à associer correctement les identificateurs avec l'élément de programme référencé.

Compilation

Une compilation est une représentation de tous les éléments nécessaires à la compilation d'un programme C# ou Visual Basic, à savoir l'ensemble des références d'assembly, des options du compilateur et des fichiers sources.

Toutes ces informations étant stockées au même emplacement, les éléments contenus dans le code source peuvent être décrits plus en détail. La compilation représente chaque élément déclaré (type, membre ou variable) sous forme de symbole. La compilation contient diverses méthodes qui vous aident à trouver et associer les symboles qui ont été déclarés dans le code source, ou importés en tant que métadonnées à partir d'un assembly.

De la même façon que les arborescences de syntaxe, les compilations sont immuables. Une fois créée, une compilation ne peut plus être modifiée, que ce soit par vous ou par tout autre utilisateur avec qui vous la partagez. Toutefois, vous pouvez créer une autre compilation à partir d'une compilation existante, en y apportant la modification souhaitée. Par exemple, vous pouvez créer une compilation semblable en tous points à une compilation existante, mais y ajouter un fichier source ou une référence d'assembly supplémentaire.

Symboles

Un symbole représente un élément distinct qui est déclaré dans le code source, ou importé en tant que métadonnées à partir d'un assembly. Chaque élément (espace de noms, type, méthode, propriété, champ, événement, paramètre ou variable locale) est représenté par un symbole.

Diverses méthodes et propriétés du type Compilation vous aident à trouver les symboles. Par exemple, vous pouvez rechercher un symbole d'un type déclaré par son nom de métadonnées commun. Vous pouvez aussi accéder à la table de symboles entière comme une arborescence de symboles ayant pour racine l'espace de noms global.

Les symboles contiennent également des informations supplémentaires que le compilateur collecte à partir du code source ou des métadonnées, par exemple, les autres symboles référencés. Chaque genre de symbole est représenté par une interface distincte dérivée de l'interface ISymbol, qui a ses propres méthodes et propriétés pour décrire en détail les informations collectées par le compilateur. Nombre de ces propriétés référencent directement d'autres symboles. Par exemple, la propriété IMethodSymbol.ReturnType indique quel symbole de type est référencé par la déclaration de méthode.

Les symboles fournissent une représentation des espaces de noms, types et membres commune au code source et aux métadonnées. Par exemple, une méthode déclarée dans le code source et une méthode importée des métadonnées sont toutes les deux représentées par un symbole IMethodSymbol ayant les mêmes propriétés.

Sur un plan conceptuel, les symboles sont similaires au système de type CLR représenté par l'API System.Reflection, mais ils sont plus complets dans la mesure où ils modélisent d'autres éléments que des types. Les espaces de noms, les variables locales et les étiquettes sont tous des symboles. De plus, les symboles sont une représentation des concepts du langage, pas des concepts du CLR. Il y a beaucoup de similitudes, mais aussi de nombreuses différences importantes. Par exemple, une méthode d'itérateur en C# ou Visual Basic est un symbole unique. Quand la méthode d'itérateur est convertie en métadonnées CLR, elle devient un type avec plusieurs méthodes.

Modèle sémantique

Un modèle sémantique représente toutes les informations sémantiques relatives à un fichier source unique. Vous pouvez utiliser ce modèle pour découvrir les éléments suivants :

- Les symboles référencés à un emplacement spécifique dans le code source.
- Le type résultant d'une expression.
- Tous les diagnostics (erreurs et avertissements).
- Les flux entrants et sortants des variables dans les régions du code source.
- Les réponses à des questions plus spéculatives.

Utiliser un espace de travail

18/03/2020 • 5 minutes to read • Edit Online

La couche des **espaces de travail** est l'emplacement à partir duquel sont effectuées l'analyse de code et la refactorisation dans les solutions entières. Dans cette couche, l'API des espaces de travail vous aide à organiser toutes les informations sur les projets d'une solution dans un modèle objet unique. Ce modèle vous permet d'accéder directement aux modèles objet de la couche du compilateur, tels que le texte source, les arborescences de syntaxe, les modèles sémantiques et les compilations, sans avoir à analyser des fichiers, configurer des options ou gérer des dépendances entre projets.

Les environnements hôtes, comme un IDE, vous fournissent un espace de travail correspondant à la solution ouverte. Vous pouvez également utiliser ce modèle en dehors d'un IDE, simplement en chargeant un fichier solution.

Espace de travail

Un espace de travail est une représentation active de la solution sous forme de collection de projets, chacun de ces projets contenant une collection de documents. Un espace de travail est généralement lié à un environnement hôte, qui change en permanence à mesure qu'un utilisateur tape ou manipule des propriétés.

La classe Workspace fournit l'accès au modèle actif de la solution. Quand un changement se produit dans l'environnement hôte, l'espace de travail déclenche les événements correspondants, et la propriété Workspace.CurrentSolution est mise à jour. Par exemple, quand un utilisateur tape dans un éditeur de texte correspondant à l'un des documents sources, l'espace de travail déclenche un événement pour signaler que le modèle global de la solution a changé et indiquer quel document a été modifié. Vous pouvez ensuite réagir à ces changements, par exemple, vérifier si le nouveau modèle est correct, surligner les zones d'intérêt ou suggérer une modification du code.

Vous pouvez également créer des espaces de travail autonomes qui sont déconnectés de l'environnement hôte ou qui sont utilisés dans une application sans environnement hôte.

Solutions, projets, documents

Même si un espace de travail change chaque fois qu'un utilisateur appuie sur une touche, vous pouvez utiliser le modèle de la solution en mode isolation.

Une solution est un modèle immuable des projets et des documents. Le modèle peut donc être partagé sans verrouillage ni duplication. Après avoir obtenu une instance de la solution à partir de la propriété Workspace.CurrentSolution, vous ne pouvez plus la changer. Toutefois, comme pour les arborescences de syntaxe et les compilations, vous pouvez modifier des solutions en créant d'autres instances basées sur des solutions existantes et en y apportant les changements souhaités. Pour répercuter vos changements dans l'espace de travail, vous devez explicitement réappliquer la solution modifiée dans l'espace de travail.

Un projet est une composante du modèle de solution immuable global. Il représente l'ensemble des documents du code source, des options d'analyse et de compilation, et des références d'assembly et entre projets. À partir d'un projet, vous pouvez accéder à la compilation correspondante sans avoir besoin de déterminer les dépendances du projet ou d'analyser des fichiers sources.

Un document est une autre composante du modèle de solution immuable global. Un document représente un fichier source unique à partir duquel vous pouvez accéder au texte du fichier, à l'arborescence de syntaxe et au modèle sémantique.

Le diagramme suivant illustre les relations entre l'espace de travail et l'environnement hôte, les outils, et la façon dont les modifications sont effectuées.

Résumé

Roslyn expose un ensemble d'API de compilateur et d'API d'espaces de travail. Ces API fournissent des informations détaillées sur votre code source et offrent une haute fidélité avec les langages Visual Basic et C#. Le SDK .NET Compiler Platform réduit considérablement les difficultés inhérentes à la création d'applications et d'outils axés sur le code. Il crée de nombreuses possibilités d'innovation dans des domaines tels que la méta-programmation, la génération et la transformation de code, l'utilisation interactive des langues de base de C et visual, et l'intégration de C et Visual Basic dans des langues spécifiques au domaine.

Explorer le code avec le visualiseur de syntaxe Roslyn dans Visual Studio

02/11/2020 • 18 minutes to read • Edit Online

Cet article fournit une vue d'ensemble de l'outil visualiseur de syntaxe inclus dans le SDK .NET Compiler Platform (« Roslyn »). Le visualiseur de syntaxe est une fenêtre Outil qui vous permet d'inspecter et d'explorer des arborescences de syntaxe. C'est un outil essentiel pour comprendre les modèles du code que vous voulez analyser. Cet outil s'avère aussi une aide précieuse au débogage quand vous développez vos propres applications à l'aide du SDK .NET Compiler Platform (« Roslyn »). Ouvrez l'outil au moment où vous créez vos premiers analyseurs. Le visualiseur facilite la compréhension des modèles utilisés par les API. Vous pouvez également utiliser des outils comme SharpLab ou LINQPad pour inspecter du code et comprendre des arborescences de syntaxe.

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Familiarisez-vous avec les concepts utilisés dans le SDK .NET Compiler Platform en lisant l'article de vue d'ensemble. Celui-ci présente les arborescences de syntaxe, les nœuds, les jetons et les trivia.

Visualiseur de syntaxe

Le **Syntax Visualizer** permet l'inspection de l'arborescence de syntaxe pour le fichier de code C# ou Visual Basic dans la fenêtre active de l'éditeur à l'intérieur de l'IDE de Visual Studio. Le visualiseur peut être lancé en cliquant sur **Afficher** > **autres fenêtres** > **Syntax Visualizer**. Vous pouvez également utiliser la barre d'outils **Lancement**

rapide dans le coin supérieur droit. Tapez « syntaxe » pour faire apparaître la commande permettant d'ouvrir le **visualiseur de syntaxe**.

Cette commande ouvre ce dernier dans une fenêtre Outil flottante. Si aucune fenêtre d'éditeur de code n'est ouverte, l'affichage est vide, comme l'illustre la figure suivante.

Ancrez cette fenêtre Outil à un emplacement pratique dans Visual Studio, comme le côté gauche. Le visualiseur indique des informations sur le fichier de code actuel.

Créez un projet à l'aide de la commande **fichier** > **nouveau projet** . Vous pouvez créer un projet Visual Basic ou C#. Quand Visual Studio ouvre le fichier de code principal de ce projet, le visualiseur en présente l'arborescence de syntaxe. Vous pouvez ouvrir n'importe quel fichier C#/Visual Basic existant dans cette instance de Visual Studio, et le visualiseur affiche l'arborescence de syntaxe de ce fichier. Si plusieurs fichiers de code sont ouverts dans Visual Studio, le visualiseur présente l'arborescence de syntaxe du fichier de code actif (celui qui a le focus clavier.)

- C#
- Visual Basic

Comme l'illustrent les images précédentes, la fenêtre Outil du visualiseur présente l'arborescence de syntaxe en haut et une grille des propriétés en bas. La grille des propriétés présente les propriétés de l'élément actuellement sélectionné dans l'arborescence, notamment le *type* .NET et le *genre* (SyntaxKind) de l'élément.

Les arborescences de syntaxe comprennent trois types d'éléments : des *nœuds*, des *jetons* et des *trivia*. Pour en savoir plus sur ces types, lisez l'article sur l'utilisation de la syntaxe. Les éléments de chaque type sont représentés à l'aide d'une couleur différente. Cliquez sur le bouton « Légende » pour obtenir une vue d'ensemble des couleurs utilisées.

Chaque élément de l'arborescence présente également sa propre **étendue**. L'**étendue** correspond aux index (position de départ et de fin) de ce nœud dans le fichier texte. Dans le précédent exemple C#, le jeton « UsingKeyword [0..5) » a une **étendue** de cinq caractères de large, [0..5). La notation « [..) » signifie que l'index de départ fait partie de l'étendue, tandis que l'index de fin n'en fait pas partie.

Il existe deux façons de naviguer dans l'arborescence :

- Développez des éléments dans l'arborescence ou cliquez dessus. Le visualiseur sélectionne automatiquement le texte correspondant à l'étendue de cet élément dans l'éditeur de code.
- Cliquez sur le texte ou sélectionnez-le dans l'éditeur de code. Dans l'exemple précédent Visual Basic, si vous sélectionnez la ligne contenant « module Module1 » dans l'éditeur de code, le visualiseur accède automatiquement au nœud ModuleStatement correspondant dans l'arborescence.

Le visualiseur met en surbrillance l'élément dans l'arborescence dont l'étendue correspond le mieux à celle du texte sélectionné dans l'éditeur.

Le visualiseur actualise l'arborescence pour qu'elle corresponde aux modifications apportées dans le fichier de code actif. Ajoutez un appel à `Console.WriteLine()` dans `Main()`. Pendant la frappe, le visualiseur actualise l'arborescence.

Interrompez la frappe une fois vous avez tapé `Console.`. L'arborescence colore certains éléments en rose. À ce stade, il existe des erreurs (également appelées « Diagnostics ») dans le code tapé. Ces erreurs sont liées à des nœuds, des jetons et des trivia dans l'arborescence de syntaxe. Le visualiseur indique les éléments auxquels sont liées des erreurs en colorant leur arrière-plan en rose. Vous pouvez inspecter les erreurs détectées sur un élément coloré en rose en plaçant le curseur au-dessus de lui. Le visualiseur montre uniquement les erreurs syntaxiques

(erreurs liées à la syntaxe du code tapé) ; il n'indique pas les erreurs sémantiques.

Graphiques de syntaxe

Cliquez avec le bouton droit sur un élément dans l'arborescence, puis cliquez sur **Afficher un graphique de syntaxe orienté**.

- C#
- Visual Basic

Le visualiseur affiche une représentation graphique de la sous-arborescence dont la racine est l'élément sélectionné. Essayez ces étapes pour le nœud **MethodDeclaration** correspondant à la méthode `Main()` dans l'exemple C#. Le visualiseur affiche un graphique de syntaxe qui ressemble à celui-ci :

La visionneuse du graphique de syntaxe contient une option permettant d'afficher une légende pour son modèle de coloration. Vous pouvez également placer le pointeur de la souris au-dessus d'éléments individuels dans le graphique de syntaxe pour afficher les propriétés correspondantes.

Vous pouvez afficher des graphiques de syntaxe pour différents éléments dans l'arborescence à plusieurs reprises. Ces graphiques s'affichent toujours dans la même fenêtre au sein de Visual Studio. Ancrez cette fenêtre à un emplacement pratique dans Visual Studio pour ne pas avoir à basculer d'un onglet à un autre afin d'afficher un nouveau graphique de syntaxe. En bas, sous les fenêtres de l'éditeur de code, s'avère souvent un emplacement pratique.

Voici la disposition d'ancrage à utiliser avec la fenêtre Outil du visualiseur et la fenêtre de graphique de syntaxe :

Une autre option consiste à placer la fenêtre de graphique de syntaxe sur un deuxième écran, dans une configuration à double écran.

Inspection de la sémantique

Le visualiseur de syntaxe permet une inspection rudimentaire des symboles et des informations sémantiques. Tapez `double x = 1 + 1;` au sein de Main() dans l'exemple C#. Ensuite, sélectionnez l'expression `1 + 1` dans la fenêtre de l'éditeur de code. Le visualiseur met en surbrillance le nœud **AddExpression** dans le visualiseur. Cliquez avec le bouton droit sur ce nœud **AddExpression**, puis cliquez sur **Afficher le symbole (le cas échéant)**. Notez que la plupart des éléments de menu présentent le qualificateur « le cas échéant ». Le visualiseur de syntaxe inspecte les propriétés d'un nœud, notamment celles qui ne sont éventuellement pas présentes pour tous les nœuds.

La grille des propriétés se met à jour dans le visualiseur comme l'illustre la figure suivante : le symbole de l'expression est un **SynthesizedIntrinsicOperatorSymbol** avec **Kind = Method**.

Essayez **Afficher TypeSymbol (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés se met à jour dans le visualiseur comme l'illustre la figure suivante, en indiquant que le type de l'expression sélectionnée est `Int32`.

Essayez **Afficher un TypeSymbol converti (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés se met à jour en indiquant que, bien que le type de l'expression soit `Int32`, le type converti de l'expression est `Double` comme l'illustre la figure suivante. Ce nœud inclut des informations de symbole de type converti, car l'expression `Int32` se produit dans un contexte où elle doit être convertie en `Double`. Cette conversion satisfait au type `Double` spécifié pour la variable `x` du côté gauche de l'opérateur d'assignation.

Enfin, essayez **Afficher la valeur constante (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés montre que la valeur de l'expression est une constante au moment de la compilation de valeur `2`.

L'exemple précédent peut également être répliqué dans Visual Basic. Tapez `Dim x As Double = 1 + 1` un fichier Visual Basic. Sélectionnez l'expression `1 + 1` dans la fenêtre de l'éditeur de code. Le visualiseur met en surbrillance le nœud **AddExpression** correspondant dans le visualiseur. Répétez les étapes précédentes pour ce nœud **AddExpression** et vous obtiendrez normalement des résultats identiques.

Examinez davantage de code dans Visual Basic. Mettez à jour votre fichier de Visual Basic principal à l'aide du code suivant :

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Ce code présente un alias nommé `C` qui se mappe sur le type `System.Console` en haut du fichier et utilise cet alias dans `Main()`. Sélectionnez l'utilisation de cet alias, le `C` dans `C.WriteLine()`, à l'intérieur de la méthode `Main()`. Le visualiseur sélectionne le nœud **IdentifierName** correspondant. Cliquez avec le bouton droit sur ce nœud et cliquez sur **Afficher le symbole (le cas échéant)**. La grille des propriétés indique que cet identificateur est lié au type `System.Console` comme l'illustre la figure suivante :

Essayez **Afficher AliasSymbol (le cas échéant)** pour le même nœud **IdentifierName**. La grille des propriétés indique que l'identificateur est un alias du nom de `C` qui est lié à la cible `System.Console`. En d'autres termes, la grille des propriétés fournit des informations sur l'**AliasSymbol** correspondant à l'identificateur `C`.

Examinez le symbole correspondant à un type, une méthode, une propriété qui ont été déclarés. Sélectionnez le nœud correspondant dans le visualiseur et cliquez sur **Afficher le symbole (le cas échéant)**. Sélectionnez la méthode `Sub Main()`, corps de la méthode compris. Cliquez sur **Afficher le symbole (le cas échéant)** pour le nœud **SubBlock** correspondant dans le visualiseur. La grille des propriétés montre que le **MethodSymbol** de ce **SubBlock** porte le nom `Main` avec un type de retour `Void`.

Les exemples de Visual Basic ci-dessus peuvent être facilement répliqués en C#. Tapez `using C = System.Console;` à la place de `Imports C = System.Console` pour l'alias. Les étapes précédentes en C# produisent des résultats identiques dans la fenêtre du visualiseur.

Les opérations d'inspection sémantique sont uniquement disponibles sur des nœuds. Elles ne le sont pas sur des jetons ni des trivia. Tous les nœuds ont des informations sémantiques intéressantes à inspecter. Quand un nœud n'a pas d'informations sémantiques intéressantes, le fait de cliquer sur **Afficher le symbole * (s'il existe)** fait apparaître une grille de propriétés vide.

Pour en savoir plus sur les API permettant d'effectuer une analyse sémantique, consultez le document de vue d'ensemble de l'utilisation de la sémantique.

Fermeture du visualiseur de syntaxe

Vous pouvez fermer la fenêtre du visualiseur quand vous ne l'utilisez pas pour examiner le code source. Le visualiseur de syntaxe met à jour son affichage au fur et à mesure que vous naviguez dans le code, en modifiant et en modifiant la source. Cela peut s'avérer perturbant quand vous ne l'utilisez pas.

Bien démarrer avec l'analyse de la syntaxe

02/11/2020 • 25 minutes to read • Edit Online

Dans ce tutoriel, vous allez explorer **l'API Syntaxe**. L'API Syntaxe fournit un accès aux structures de données qui décrivent un programme C# ou Visual Basic. Ces structures de données ont suffisamment d'informations détaillées pour pouvoir représenter entièrement n'importe quel programme, quelle que soit sa taille. Ces structures peuvent décrire des programmes complets qui se compilent et s'exécutent correctement. Elles peuvent aussi décrire des programmes incomplets, au fil de leur écriture dans l'éditeur.

Pour permettre cette expression détaillée, les structures de données et les API qui composent l'API Syntaxe sont nécessairement complexes. Commençons par ce à quoi ressemble la structure de données pour le programme habituel « Hello World » :

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Considérez le texte du programme précédent. Vous reconnaissez des éléments familiers. Le texte entier représente un même fichier source, ou une **unité de compilation**. Les trois premières lignes de ce fichier source sont des **directives using**. Le reste de la source est contenue dans une **déclaration d'espace de noms**. La déclaration d'espace de noms contient une **déclaration de classe** enfant. La déclaration de classe contient une **déclaration de méthode**.

L'API Syntaxe crée une arborescence dont la racine représente l'unité de compilation. Les nœuds de l'arborescence représentent les directives using, la déclaration d'espace de noms et tous les autres éléments du programme. L' arborescence continue jusqu'aux niveaux les plus bas : la chaîne « Hello World! » est un **jeton de littéral de chaîne** qui est un descendant d'un **argument**. L'API Syntaxe fournit l'accès à la structure du programme. Vous pouvez rechercher des utilisations spécifiques du code, parcourir toute l'arborescence pour comprendre le code et créer des arborescences en modifiant l'arborescence existante.

Cette brève description fournit une vue d'ensemble des types d'informations accessibles avec l'API Syntaxe. L'API Syntaxe n'est rien de plus qu'une API formelle qui décrit les constructions de code familières que vous connaissez en C#. Les fonctionnalités complètes incluent des informations sur la façon dont le code est mis en forme, notamment les sauts de ligne, les espaces et l'indentation. Avec ces informations, vous pouvez représenter entièrement le code comme étant écrit et lu par des programmeurs ou par le compilateur. L'utilisation de cette structure vous permet d'interagir avec le code source à un niveau qui est vraiment significatif. Il ne s'agit plus de chaînes de texte, mais de données qui représentent la structure d'un programme C#.

Pour commencer, vous devez installer le kit **.NET Compiler Platform SDK** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Présentation des arborescences de syntaxe

Vous utilisez l'API Syntaxe pour les analyses de la structure du code C#. **L'API Syntaxe** expose les analyseurs, les arborescences de syntaxe, et des utilitaires pour l'analyse et la construction d'arborescences de syntaxe. C'est ainsi que vous recherchez le code pour des éléments de syntaxe spécifiques ou que vous lisez le code d'un programme.

Une arborescence de syntaxe est une structure de données utilisée par les compilateurs C# et Visual Basic pour comprendre les programmes C# et Visual Basic. Les arborescences de syntaxe sont produites par le même analyseur que celui qui s'exécute quand un projet est généré ou qu'un développeur appuie sur F5. Les arborescences de syntaxe sont parfaitement fidèles au langage : chaque élément d'information d'un fichier de code est représenté dans l'arborescence. L'écriture d'une arborescence de syntaxe en texte reproduit le texte d'origine exact qui a été analysé. Les arborescences de syntaxe sont également **immuables** : une fois créée, une arborescence de syntaxe ne peut jamais être modifiée. Les consommateurs des arborescences peuvent les analyser sur plusieurs threads, sans verrous ou d'autres mesures liées à la simultanéité des accès, sachant que les données ne changent jamais. Vous pouvez utiliser des API pour créer des arborescences qui sont le résultat de la modification d'une arborescence existante.

Les quatre principaux blocs de construction des arborescences de syntaxe sont :

- La classe Microsoft.CodeAnalysis.SyntaxTree, une instance de celle-ci représentant une arborescence d'analyse entière. SyntaxTree est une classe abstraite avec des dérivés spécifiques à un langage. Vous utilisez les méthodes parse de la Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree classe (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) pour analyser le texte en C# (ou Visual Basic).
- La classe Microsoft.CodeAnalysis.SyntaxNode, dont les instances représentent des constructions syntaxiques, comme des déclarations, des instructions, des clauses et des expressions.
- La structure Microsoft.CodeAnalysis.SyntaxToken, qui représente un mot clé, un identificateur, un opérateur ou un signe de ponctuation individuel.

- Enfin, la structure Microsoft.CodeAnalysis.SyntaxTrivia, qui représente les éléments d'information non significatifs du point de vue syntaxique, comme l'espace entre des jetons, des directives de prétraitement et des commentaires.

Tout cela, les jetons et les nœuds sont composés de façon hiérarchique pour former une arborescence représentant complètement tous les éléments d'un fragment de code Visual Basic ou C#. Vous pouvez voir cette structure en utilisant la fenêtre **Visualiseur de syntaxe**. Dans Visual Studio, choisissez **Afficher** > **les autres** > **Syntax Visualizer**Windows. Par exemple, le fichier source C# précédent examiné avec le **Visualiseur de syntaxe** se présente comme dans la figure suivante :

SyntaxNode : Bleu | **SyntaxToken** : Vert | **SyntaxTrivia** : Rouge

En parcourant cette arborescence, vous trouverez les instructions, expressions, jetons ou espaces, quels qu'ils soient, dans un fichier de code.

Vous pouvez rechercher n'importe quel élément dans un fichier de code à l'aide des API Syntaxe, mais la plupart des scénarios concernent l'examen de petits extraits de code, ou la recherche d'instructions ou de fragments particuliers. Les deux exemples qui suivent montrent des utilisations classiques d'exploration de la structure du code ou de recherche d'instructions individuelles.

Exploration des arborescences

Vous pouvez examiner les nœuds d'une arborescence de syntaxe de deux façons. Vous pouvez parcourir l'arborescence pour examiner chaque nœud, ou vous pouvez rechercher des éléments ou des nœuds spécifiques.

Exploration manuelle

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub.

> **NOTE**
> Les types de l'arborescence de syntaxe utilisent l'héritage pour décrire les différents éléments de syntaxe qui sont valides à différents emplacements du programme. Utiliser ces API signifie souvent effectuer un cast des propriétés ou des membres de collection vers des types dérivés spécifiques. Dans les exemples suivants, l'affectation et les casts sont des instructions distinctes, qui utilisent des variables typées explicitement. Vous pouvez lire le code pour voir les types de retour de l'API et le type à l'exécution des objets retournés. Dans la pratique, il est plus courant d'utiliser des variables typées implicitement et de se baser sur des noms d'API pour décrire le type des objets examinés.

Créez un projet C# **Outil d'analyse du code autonome** :

- Dans Visual Studio, choisissez **fichier** > **nouveau** > **projet** pour afficher la boîte de dialogue Nouveau projet.
- Sous **extensibilité Visual C#** > **Extensibility**, choisissez **outil d'analyse du code autonome**.
- Nommez votre projet « **SyntaxTreeManualTraversal** » et cliquez sur OK.

Vous allez analyser le programme simple « Hello World! » montré précédemment. Ajoutez le texte pour le programme Hello World en tant que constante dans votre classe `Program` :

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Ensuite, ajoutez le code suivant pour générer **l'arborescence de syntaxe** pour le texte du code dans la constante `programText`. Ajoutez la ligne suivante à votre méthode `Main` :

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ces deux lignes créent l'arborescence et en récupèrent le nœud racine. Vous pouvez maintenant examiner les nœuds de l'arborescence. Ajoutez ces lignes à votre méthode `Main` pour afficher certaines des propriétés du nœud racine de l'arborescence :

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Exécutez l'application pour voir ce que votre code a découvert sur le nœud racine de cette arborescence.

En règle générale, vous explorez l'arborescence pour en savoir plus sur le code. Dans cet exemple, vous analysez du code que vous connaissez pour explorer les API. Ajoutez le code suivant pour examiner le premier membre du nœud `root` :

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Ce membre est une Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Il représente tous les éléments de l'étendue de la déclaration `namespace HelloWorld`. Ajoutez le code suivant pour examiner les nœuds qui sont déclarés à l'intérieur de l'espace de noms `HelloWorld` :

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in this namespace.");
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Exécutez le programme pour voir ce que vous avez découvert.

Maintenant que vous savez que la déclaration est une Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, déclarez une nouvelle variable de ce type pour examiner la déclaration de la classe. Cette classe contient seulement un membre : la méthode `Main`. Ajoutez le code suivant pour rechercher la méthode `Main` et en effectuer un cast en Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

Le nœud de la déclaration de la méthode contient toutes les informations syntaxiques relatives à la méthode. Affichons le type de retour de la méthode `Main`, le nombre et les types des arguments, et le texte du corps de la méthode. Ajoutez le code suivant :

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is {mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count} parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Exécutez le programme pour voir toutes les informations que vous avez découvertes sur ce programme :

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Méthodes de requête

En plus de parcourir les arborescences, vous pouvez aussi explorer l'arborescence de syntaxe avec les méthodes de requête définies sur Microsoft.CodeAnalysis.SyntaxNode. Ces méthodes sont normalement immédiatement familières à toute personne connaissant XPath. Vous pouvez utiliser ces méthodes avec LINQ pour rechercher rapidement des éléments dans une arborescence. SyntaxNode a des méthodes de requête comme DescendantNodes, AncestorsAndSelf et ChildNodes.

Vous pouvez utiliser ces méthodes de requête pour rechercher l'argument de la méthode `Main`, au lieu de naviguer dans l'arborescence. Ajoutez le code suivant dans le bas de votre nouvelle méthode `Main` :

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

La première instruction utilise une expression LINQ et la méthode DescendantNodes pour localiser le même paramètre que dans l'exemple précédent.

Exécutez le programme : vous voyez que l'expression LINQ a trouvé le même paramètre que quand vous avez exploré manuellement l'arborescence.

L'exemple utilise des instructions `WriteLine` pour afficher des informations sur les arborescences de syntaxe au fil de leur exploration. Vous pouvez aussi découvrir beaucoup plus d'informations en exécutant le programme sous le débogueur. Vous pouvez examiner plus de propriétés et de méthodes qui font partie de l'arborescence de syntaxe créée pour le programme « Hello World ».

Parcoureurs de syntaxe

Vous voulez souvent rechercher tous les nœuds d'un type spécifique dans une arborescence de syntaxe, par exemple toutes les déclarations de propriété dans un fichier. En étendant la classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker et en remplaçant la méthode VisitPropertyDeclaration(PropertyDeclarationSyntax), vous traitez chaque déclaration de propriété d'une arborescence de syntaxe sans connaître préalablement sa structure. CSharpSyntaxWalker est un type spécifique de CSharpSyntaxVisitor, qui visite de façon récursive un nœud et chacun de ses enfants.

Cet exemple implémente un CSharpSyntaxWalker qui examine une arborescence de syntaxe. Il collecte les directives `using` qu'il trouve et qui n'importent pas un espace de noms `System`.

Créez un projet C# **Outil d'analyse du code autonome** et nommez-le « **SyntaxWalker** ».

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub. L'exemple sur GitHub contient les deux projets décrits dans ce tutoriel.

Comme dans l'exemple précédent, vous pouvez définir une constante de chaîne pour contenir le texte du programme que vous voulez analyser :

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

Ce texte source contient des directives `using` réparties dans quatre emplacements différents : le niveau fichier, dans l'espace de noms du plus haut niveau et dans les deux espaces de noms imbriqués. Cet exemple met en évidence un scénario de base pour l'utilisation de la classe CSharpSyntaxWalker pour interroger du code. Il serait fastidieux de visiter chaque nœud dans l'arborescence de syntaxe racine pour trouver les déclarations using. Au lieu de cela, vous créez une classe dérivée et vous remplacez la méthode qui est appelée seulement quand le nœud actuel de l'arborescence est une directive using. Votre visiteur n'effectue aucun travail sur les autres types de nœuds. Cette même méthode examine chacune des instructions `using` et génère une collection des espaces de noms qui ne figurent pas dans l'espace de noms `System`. Vous créez un CSharpSyntaxWalker qui examine toutes les instructions `using`, mais uniquement les instructions `using`.

Maintenant que vous avez défini le texte du programme, vous devez créer un `SyntaxTree` et obtenir la racine de cette arborescence :

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ensuite, créez une classe. Dans Visual Studio, choisissez **projet** > **Ajouter un nouvel élément**. Dans la boîte de dialogue **Ajouter un nouvel élément**, tapez *UsingCollector.cs* comme nom de fichier.

Vous implémentez la fonctionnalité de visiteur `using` dans la classe `UsingCollector`. Commencez par faire de la classe `UsingCollector` une dérivée CSharpSyntaxWalker.

```
class UsingCollector : CSharpSyntaxWalker
```

Vous avez besoin d'un stockage pour contenir les nœuds d'espace de noms que vous collectez. Déclarez une propriété publique en lecture seule dans la classe `UsingCollector` ; vous utilisez cette variable pour stocker les nœuds UsingDirectiveSyntax que vous trouvez :

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();
```

La classe de base, CSharpSyntaxWalker, implémente la logique pour visiter chaque nœud de l'arborescence de syntaxe. La classe dérivée remplace les méthodes appelées pour les nœuds spécifiques qui vous intéressent. Dans ce cas, vous êtes intéressé par toutes les directives `using`. Cela signifie que vous devez remplacer la méthode VisitUsingDirective(UsingDirectiveSyntax). Le seul argument de cette méthode est un objet Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax. Utiliser des visiteurs a un avantage important : ils appellent les méthodes remplacées avec des arguments déjà castés au type de nœud spécifique. La classe Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax a une propriété Name qui stocke le nom de l'espace de noms importé. Il s'agit d'une Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Ajoutez le code suivant dans le remplacement de VisitUsingDirective(UsingDirectiveSyntax) :

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Comme avec l'exemple précédent, vous avez ajouté différentes instructions `WriteLine` pour faciliter la compréhension de cette méthode. Vous pouvez voir quand elle est appelée, ainsi que les arguments qui lui sont passés à chaque fois.

Enfin, vous devez ajouter deux lignes de code pour créer le `UsingCollector` et faire en sorte qu'il visite le nœud racine, en collectant toutes les instructions `using`. Ensuite, ajoutez une boucle `foreach` pour afficher toutes les instructions `using` trouvées par votre collecteur :

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compilez et exécutez le programme. Vous devez normalement voir la sortie suivante :

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Félicitations ! Vous avez utilisé **l'API Syntaxe** pour rechercher des types spécifiques d'instructions et de déclarations C# dans du code source C#.

Bien démarrer avec l'analyse sémantique

18/03/2020 • 16 minutes to read • Edit Online

Ce tutoriel suppose que vous êtes familiarisé avec l'API Syntaxe. L'article Bien démarrer avec l'analyse syntaxique fournit une introduction suffisante.

Dans ce tutoriel, vous découvrez les API **Symbole** et **Liaison**. Ces API fournissent des informations sur la *signification sémantique* d'un programme. Elles vous permettent de poser et de répondre à des questions sur les types représentés par un symbole dans votre programme.

Vous devez installer le SDK **.NET Compiler Platform** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Présentation des compilations et des symboles

Au fil de votre utilisation du SDK du compilateur .NET, vous allez mieux comprendre les différences entre l'API Syntaxe et l'API Sémantique. **L'API Syntaxe** vous permet d'examiner la *structure* d'un programme. Cependant, des informations plus détaillées sur la sémantique ou la *signification* d'un programme sont souvent nécessaires. Bien qu'un fichier de code lâche ou un extrait de code Visual Basic ou Cmd puisse être analysé de façon syntaxique de façon isolée, il n'est pas significatif de poser des questions telles que « quel est le type de cette variable » dans le vide. La signification d'un nom de type peut dépendre de références d'assembly, d'importations d'espaces de noms ou d'autres fichiers de code. Ces questions sont traitées avec **l'API Sémantique**, plus précisément avec la classe

Microsoft.CodeAnalysis.Compilation.

Une instance de Compilation est analogue à un projet tel qu'il est vu par le compilateur et représente tout ce qui est nécessaire pour compiler un programme Visual Basic ou C#. La **compilation** comprend l'ensemble des fichiers sources à compiler, les références d'assembly et les options du compilateur. Vous pouvez analyser la signification du code en utilisant toutes les autres informations de ce contexte. Une Compilation vous permet de rechercher des **symboles**, qui sont des entités comme des types, des espaces de noms, des membres et des variables, qui sont référencées par des noms et d'autres expressions. Le processus consistant à associer des noms et des expressions à des **symboles** est appelé la **liaison**.

Comme Microsoft.CodeAnalysis.SyntaxTree, Compilation est une classe abstraite avec des dérivés spécifiques à un langage. Quand vous créez une instance de compilation, vous devez appeler une méthode de fabrique sur la classe Microsoft.CodeAnalysis.CSharp.CSharpCompilation (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Interrogation des symboles

Dans ce tutoriel, vous examinez à nouveau le programme « Hello World ». Cette fois-ci, vous interrogez les symboles du programme pour comprendre quels types ces symboles représentent. Vous interrogez les types dans un espace de noms et vous découvrez comment trouver les méthodes disponibles sur un type.

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub.

> **NOTE**
>
> Les types de l'arborescence de syntaxe utilisent l'héritage pour décrire les différents éléments de syntaxe qui sont valides à différents emplacements du programme. Utiliser ces API signifie souvent effectuer un cast des propriétés ou des membres de collection vers des types dérivés spécifiques. Dans les exemples suivants, l'affectation et les casts sont des instructions distinctes, qui utilisent des variables typées explicitement. Vous pouvez lire le code pour voir les types de retour de l'API et le type à l'exécution des objets retournés. Dans la pratique, il est plus courant d'utiliser des variables typées implicitement et de se baser sur des noms d'API pour décrire le type des objets examinés.

Créez un projet C# **Outil d'analyse du code autonome** :

- Dans Visual Studio, choisissez **File** > **New** > **Project** pour afficher le dialogue du nouveau projet.
- Sous Visual C **'Extensibility Visual C#** > , choisissez **Un outil d'analyse de code autonome**.
- Nommez votre projet « **SemanticQuickStart** » et cliquez sur OK.

Vous allez analyser le programme simple « Hello World! » montré précédemment. Ajoutez le texte pour le programme Hello World en tant que constante dans votre classe `Program` :

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Ensuite, ajoutez le code suivant pour générer l'arborescence de syntaxe pour le texte du code dans la constante `programText`. Ajoutez la ligne suivante à votre méthode `Main` :

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ensuite, générez une CSharpCompilation à partir de l'arborescence que vous avez déjà créée. L'exemple « Hello World » est basé sur les types String et Console. Vous devez référencer l'assembly qui déclare ces deux types dans votre compilation. Ajoutez la ligne suivante à votre méthode `Main` pour créer une compilation de votre arborescence de syntaxe, en incluant la référence à l'assembly approprié :

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

La méthode CSharpCompilation.AddReferences ajoute des références à la compilation. La méthode MetadataReference.CreateFromFile charge un assembly en tant que référence.

Interrogation du modèle sémantique

Une fois que vous avez une Compilation, vous pouvez la demander pour un SemanticModel pour n'importe quelle SyntaxTree contenue dans Compilation. Vous pouvez considérer le modèle sémantique comme étant la source de toutes les informations que vous obtiendriez normalement d'IntelliSense. Un SemanticModel peut répondre à des questions comme « Quels noms se trouvent dans l'étendue à cet emplacement ? », « Quels membres sont accessibles à partir de cette méthode ? », « Quelles variables sont utilisées dans ce bloc de texte ? » et « À quoi ce nom ou cette expression font-ils référence ? ». Ajoutez cette instruction pour créer le modèle sémantique :

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Liaison d'un nom

Compilation crée SemanticModel à partir de SyntaxTree. Après avoir créé le modèle, vous pouvez l'interroger pour rechercher la première directive `using` et récupérer les informations des symboles pour l'espace de noms `System`. Ajoutez ces deux lignes à votre méthode `Main` pour créer le modèle sémantique et pour récupérer le symbole pour la première instruction using :

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

Le code précédent montre comment lier le nom dans la première directive `using` pour récupérer un Microsoft.CodeAnalysis.SymbolInfo pour l'espace de noms `System`. Le code précédent montre aussi que vous utilisez le **modèle de syntaxe** pour trouver la structure du code ; vous utilisez le **modèle sémantique** pour comprendre sa signification. Le **modèle de syntaxe** recherche la chaîne `System` dans l'instruction using. Le **modèle sémantique** a toutes les informations sur les types définis dans l'espace de noms `System`.

À partir de l'objet SymbolInfo, vous pouvez obtenir Microsoft.CodeAnalysis.ISymbol en utilisant la propriété SymbolInfo.Symbol. Cette propriété retourne le symbole auquel cette expression fait référence. Pour les

expressions qui ne font référence à rien (comme les littéraux numériques), cette propriété est `null`. Quand SymbolInfo.Symbol n'est pas null, ISymbol.Kind indique le type du symbole. Dans cet exemple, la propriété ISymbol.Kind est un SymbolKind.Namespace. Ajoutez le code suivant à votre méthode `Main`. Il récupère le symbole pour l'espace de noms `System`, puis affiche tous les espaces de noms enfants déclarés dans l'espace de noms `System` :

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Exécutez le programme ; vous devez normalement voir la sortie suivante :

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

> **NOTE**
> La sortie n'inclut pas chaque espace de noms qui est un espace de noms enfant de l'espace de noms `System`. Elle affiche chaque espace de noms qui est présent dans cette compilation, qui référence seulement l'assembly où `System.String` est déclaré. Les espaces de noms déclarés dans d'autres assemblys ne sont pas connus de cette compilation.

Liaison d'une expression

Le code précédent montre comment trouver un symbole en le liant à un nom. D'autres expressions existent dans un programme C# qui peuvent être liées, mais qui ne sont pas des noms. Pour montrer cette possibilité, accédons à la liaison vers un littéral de chaîne simple.

Le programme « Hello World » contient un Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, : la chaîne « Hello, World! » affichée dans la console.

Vous trouvez la chaîne « Hello, World! » en recherchant le seul littéral de chaîne dans le programme. Ensuite, une fois que vous avez localisé le nœud de syntaxe, vous obtenez les informations de type pour ce nœud auprès du modèle sémantique. Ajoutez le code suivant à votre méthode `Main` :

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

La struct Microsoft.CodeAnalysis.TypeInfo inclut une propriété TypeInfo.Type qui permet d'accéder aux

informations sémantiques sur le type du littéral. Dans cet exemple, il s'agit du type `string`. Ajoutez une déclaration qui affecte cette propriété à une variable locale :

```
var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;
```

Pour terminer ce tutoriel, construisons une requête LINQ qui crée une séquence de toutes les méthodes publiques déclarées sur le type `string` qui retournent un type `string`. Cette requête est complexe : nous allons donc la construire ligne par ligne, puis la reconstruire pour former une seule requête. La source de cette requête est la séquence de tous les membres déclarés sur le type `string` :

```
var allMembers = stringTypeSymbol.GetMembers();
```

Cette séquence source contient tous les membres, notamment les propriétés et les champs ; filtrez-la avec la méthode ImmutableArray<T>.OfType pour trouver les éléments qui sont des objets Microsoft.CodeAnalysis.IMethodSymbol :

```
var methods = allMembers.OfType<IMethodSymbol>();
```

Ensuite, ajoutez un autre filtre pour retourner seulement les méthodes qui sont publiques et qui retournent un type `string` :

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Sélectionnez seulement la propriété name et seulement les noms distincts en supprimant les surcharges :

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

Vous pouvez aussi créer toute la requête avec la syntaxe de requête LINQ, puis afficher tous les noms de méthode dans la console :

```
foreach (string name in (from method in stringTypeSymbol
                         .GetMembers().OfType<IMethodSymbol>()
                         where method.ReturnType.Equals(stringTypeSymbol) &&
                         method.DeclaredAccessibility == Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Générez et exécutez le programme. Vous devez normalement voir la sortie suivante.

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

Vous avez utilisé l'API Sémantique pour rechercher et afficher des informations sur les symboles qui font partie de ce programme.

Bien démarrer avec la transformation de la syntaxe

02/11/2020 • 24 minutes to read • Edit Online

Ce tutoriel s'appuie sur les concepts et techniques décrits dans les démarrages rapides Bien démarrer avec l'analyse de la syntaxe et Bien démarrer avec l'analyse sémantique. Si vous ne l'avez pas déjà fait, vous devez terminer ces démarrages rapides avant de commencer celui-ci.

Dans ce démarrage rapide, vous découvrez les techniques de création et de transformation des arborescences de syntaxe. En association avec les techniques que vous avez apprises dans les démarrages rapides précédents, vous créez votre première refactorisation de ligne de commande !

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Immuabilité et plateforme de compilateur .NET

L'**immuabilité** est un principe fondamental de la plateforme de compilateur .NET. Une fois créées, les structures de données immuables ne peuvent plus être modifiées. Les structures de données immuables peuvent être partagées et analysées simultanément par plusieurs consommateurs, en toute sécurité. Il n'existe aucun risque qu'un consommateur n'en perturbe un autre de manière imprévisible. Votre analyseur n'a pas besoin de verrous ou d'autres mesures de concurrence. Cette règle s'applique aux arborescences de syntaxe, compilations, symboles, modèles sémantiques et toute autre structure de données que vous rencontrez. Au lieu de modifier des structures existantes, les API créent de nouveaux objets en fonction des différences spécifiés dans les anciens objets. Vous appliquez ce concept aux arborescences de syntaxe pour créer de nouvelles arborescences à l'aide de

transformations.

Créer et transformer des arborescences

Vous choisissez une des deux stratégies pour les transformations de syntaxe. Les **méthodes de fabrique** sont particulièrement utiles lorsque vous recherchez des nœuds spécifiques à remplacer, ou des emplacements spécifiques dans lesquelles vous souhaitez insérer le nouveau code. Les **modules de réécriture** sont recommandés lorsque vous souhaitez analyser un projet complet et y rechercher des modèles de code à remplacer.

Créer des nœuds avec des méthodes de fabrique

La première transformation de syntaxe montre les méthodes de fabrique. Vous allez remplacer une instruction `using System.Collections;` par une instruction `using System.Collections.Generic;`. Cet exemple montre comment créer des objets Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode à l'aide des méthodes de fabrique Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Pour chaque type de **nœud**, **jeton** ou **trivia**, il existe une méthode de fabrique qui crée une instance de ce type. Vous créez des arborescences de syntaxe en composant hiérarchiquement des nœuds de façon ascendante. Puis vous transformez le programme existant en remplaçant les nœuds existants par la nouvelle arborescence que vous avez créée.

Démarrez Visual Studio, puis créez un nouveau projet C# **Outil d'analyse du code autonome**. Dans Visual Studio, choisissez **fichier** > **nouveau** > **projet** pour afficher la boîte de dialogue Nouveau projet. Sous **extensibilité Visual C#** > **Extensibility** , choisissez un **outil d'analyse du code autonome**. Ce démarrage rapide inclut deux exemples de projets, par conséquent, nommez la solution **SyntaxTransformationQuickStart**, puis nommez le projet **ConstructionCS**. Cliquez sur **OK**.

Ce projet utilise les méthodes de classe Microsoft.CodeAnalysis.CSharp.SyntaxFactory pour construire un élément Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax représentant l'espace de noms `System.Collections.Generic`.

Ajoutez la directive using suivante au début de `Program.cs` .

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Vous allez créer des **nœuds de syntaxe du nom** pour générer l'arborescence qui représente l'instruction `using System.Collections.Generic;`. NameSyntax est la classe de base pour les quatre types de noms qui s'affichent en C#. Vous composez ces quatre types de noms à la fois pour créer n'importe quel nom pouvant s'afficher en langage C# :

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, qui représente des noms d'identificateur unique simples comme `System` et `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, qui représente un nom de type ou de méthode générique comme `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax, qui représente un nom qualifié de formulaire `<left-name>.<right-identifier-or-generic-name>` comme `System.IO`.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, qui représente un nom en utilisant un alias d'assembly externe comme `LibraryV2::Foo`.

Vous utilisez la méthode IdentifierName(String) pour créer un nœud NameSyntax. Ajoutez le code suivant à votre méthode `Main`, dans `Program.cs` :

```
NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

Le code précédent crée un objet IdentifierNameSyntax et l'assigne à la variable `name`. La plupart des API Roslyn

renvoient des classes de base pour les rendre plus faciles à utiliser avec les types associés. La variable `name`, une NameSyntax, peut être réutilisée lorsque vous générez la QualifiedNameSyntax. N'utilisez pas l'inférence de type lorsque vous générez l'exemple. Vous allez automatiser cette étape dans ce projet.

Vous avez créé le nom. Maintenant, il est temps de créer d'autres nœuds dans l'arborescence en générant une QualifiedNameSyntax. La nouvelle arborescence utilise `name` comme partie gauche du nom et une nouvelle IdentifierNameSyntax pour l'espace de noms `Collections` comme partie droite de la QualifiedNameSyntax. Ajoutez le code suivant à `program.cs` :

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Exécutez à nouveau le code et affichez les résultats. Vous créez une arborescence de nœuds qui représente le code. Vous allez continuer à utiliser ce modèle pour générer la QualifiedNameSyntax pour l'espace de noms `System.Collections.Generic`. Ajoutez le code suivant à `Program.cs` :

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Réexécutez le programme pour voir que vous avez créé l'arborescence pour le code à ajouter.

Créer une arborescence modifiée

Vous avez créé petite arborescence de syntaxe contenant une instruction. Les API permettant de créer de nouveaux nœuds représentent la solution idéale pour créer des instructions uniques ou d'autres petits blocs de code. Toutefois, pour générer de plus grands blocs de code, vous devez utiliser des méthodes qui remplacent les nœuds ou insèrent des nœuds dans une arborescence existante. N'oubliez pas que les arborescences de syntaxe sont immuables. L'**API Syntaxe** ne fournit aucun mécanisme permettant de modifier une arborescence de syntaxe existante après la construction. Au lieu de cela, elle fournit des méthodes qui génèrent de nouvelles arborescences en fonction des modifications apportées aux arborescences existantes. Les méthodes `With*` sont définies dans des classes concrètes qui dérivent de SyntaxNode ou dans des méthodes d'extension déclarées dans la classe SyntaxNodeExtensions. Ces méthodes créent un nouveau nœud en appliquant les modifications apportées aux propriétés enfants d'un nœud existant. En outre, la méthode d'extension ReplaceNode peut être utilisée pour remplacer un nœud descendant dans une sous-arborescence. Cette méthode met également à jour le parent pour pointer vers l'enfant qui vient d'être créé, et répète ce processus pour l'arborescence complète. Ce processus est appelé « *re-spining* » d'arborescence.

L'étape suivante consiste à créer une arborescence qui représente un (petit) programme complet puis à le modifier. Ajoutez le code suivant au début de la classe `Program` :

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> **NOTE**
> L'exemple de code utilise l'espace de noms `System.Collections` et non pas l'espace de noms `System.Collections.Generic`.

Ensuite, ajoutez le code suivant en bas de la méthode `Main` pour analyser le texte et créer une arborescence :

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Cet exemple utilise la méthode WithName(NameSyntax) pour remplacer le nom dans un nœud UsingDirectiveSyntax par celui construit dans le code précédent.

Créez un nouveau nœud UsingDirectiveSyntax à l'aide de la méthode WithName(NameSyntax) pour mettre à jour le nom `System.Collections` avec le nom que vous avez créé dans le code précédent. Ajoutez le code suivant dans le bas de la méthode `Main` :

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Exécutez le programme et examinez attentivement la sortie. `newusing` n'a pas été placé dans l'arborescence racine. L'arborescence d'origine n'a pas été modifiée.

Ajoutez le code suivant à l'aide de la méthode d'extension ReplaceNode pour créer une nouvelle arborescence. La nouvelle arborescence est le résultat du remplacement de l'importation existante par le nœud `newUsing` mis à jour. Vous affectez cette nouvelle arborescence à la variable `root` existante :

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Réexécutez le programme. Cette fois, l'arborescence importe correctement l'espace de noms `System.Collections.Generic`.

Transformer des arborescences avec `SyntaxRewriters`

Les méthodes `With*` et ReplaceNode constituent une solution pratique pour transformer des branches individuelles en une arborescence de syntaxe. La classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter effectue plusieurs transformations sur une arborescence de syntaxe. La classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter est une sous-classe de Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. CSharpSyntaxRewriter applique une transformation à un type spécifique de SyntaxNode. Vous pouvez appliquer des transformations à plusieurs types d'objets SyntaxNode là où elles apparaissent dans une arborescence de syntaxe. Le second projet de ce démarrage rapide crée une refactorisation de ligne de commande qui supprime des types explicites dans des déclarations de variables locales partout où une inférence de type peut être utilisée.

Créez un nouveau projet C# d' **outil d'analyse du code** autonome. Dans Visual Studio, cliquez avec le bouton droit sur le nœud de la solution `SyntaxTransformationQuickStart`. Choisissez **Ajouter** > **un nouveau projet** pour afficher la **boîte de dialogue Nouveau projet**. Sous **extensibilité Visual C#** > **Extensibility**, choisissez **outil d'analyse du code autonome**. Nommez votre projet `TransformationCS`, puis cliquez sur OK.

La première étape consiste à créer une classe dérivée de CSharpSyntaxRewriter pour effectuer vos transformations. Ajoutez un nouveau fichier de classe au projet. Dans Visual Studio, choisissez **projet** > **Ajouter une classe...**. Dans

la boîte de dialogue **Ajouter un nouvel élément** `TypeInferenceRewriter.cs` , tapez comme nom de fichier.

Ajoutez le code suivant à l'aide de directives dans le fichier `TypeInferenceRewriter.cs` :

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Ensuite, étendez la classe `TypeInferenceRewriter` à la classe CSharpSyntaxRewriter :

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Ajoutez le code suivant pour déclarer un champ en lecture seule privé qui contiendra une SemanticModel et initialisez-le dans le constructeur. Vous aurez besoin de ce champ ultérieurement pour déterminer où l'inférence de type peut être utilisée :

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Remplacez la méthode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) :

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> **NOTE**
> De nombreuses API Roslyn déclarent des types de retour représentant des classes de base des types Runtime réels retournés. Dans de nombreux scénarios, il est possible de remplacer un type de nœud par un autre, voire de le supprimer. Dans cet exemple, la méthode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) retourne un SyntaxNode, au lieu du type dérivé de LocalDeclarationStatementSyntax. Ce module de réécriture retourne un nouveau nœud LocalDeclarationStatementSyntax en fonction du nœud existant.

Ce démarrage rapide gère les déclarations de variables locales. Vous pouvez l'étendre à d'autres déclarations telles que des boucles `foreach`, des boucles `for`, des expressions LINQ et des expressions lambda. En outre, ce module de réécriture transformera les déclarations dans la forme la plus simple :

```
Type variable = expression;
```

Si vous souhaitez explorer votre propre déclaration, vous pouvez étendre l'exemple terminé pour ces types de déclarations de variables :

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Ajoutez le code suivant au corps de la méthode `VisitLocalDeclarationStatement` pour ignorer la réécriture de ces formes de déclarations :

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

La méthode indique qu'aucune réécriture n'a lieu en retournant le paramètre `node` inchangé. Si aucune de ces expressions `if` n'est vraie, le nœud représente une déclaration possible avec initialisation. Ajoutez les instructions suivantes pour extraire le nom du type spécifié dans la déclaration et liez-le à l'aide du champ SemanticModel pour obtenir un symbole de type :

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Ajoutez maintenant cette instruction pour lier l'expression de l'initialiseur :

```
var initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Enfin, ajoutez l'instruction `if` suivante pour remplacer le nom de type existant par le mot clé `var` si le type de l'expression de l'initialiseur correspond au type spécifié :

```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

Cette condition est nécessaire car la déclaration peut effectuer un cast de l'expression de l'initialiseur dans une classe de base ou une interface. Si vous le souhaitez, les types à gauche et à droite de l'affectation ne correspondent pas. Dans ce cas, la suppression du type explicite modifierait la sémantique d'un programme. `var` est spécifié comme un identificateur plutôt qu'un mot clé car `var` est un mot clé contextuel. Les trivias de début et de fin (espaces blancs) sont transférés de l'ancien nom de type vers le mot clé `var` afin de conserver les espaces blancs verticaux et l'indentation. Il est plus simple d'utiliser `ReplaceNode` plutôt que `With*` pour transformer la LocalDeclarationStatementSyntax, car le nom de type est en fait le petit-enfant de l'instruction de déclaration.

Vous avez terminé le `TypeInferenceRewriter`. Revenez maintenant à votre fichier `Program.cs` pour terminer l'exemple. Créez un test Compilation et obtenez le SemanticModel à partir de celui-ci. Utilisez SemanticModel pour tester votre `TypeInferenceRewriter`. Vous effectuerez cette étape en dernier. En attendant, déclarez une variable d'espace réservé représentant votre compilation de test :

```
Compilation test = CreateTestCompilation();
```

Après une pause, vous devriez voir apparaître une erreur sous forme de tilde, indiquant qu'il n'existe aucune méthode `CreateTestCompilation`. Appuyez sur **Ctrl+point** pour ouvrir l'ampoule, puis sur Entrée pour appeler la commande **Générer un stub de méthode**. Cette commande va générer un stub de méthode pour la méthode `CreateTestCompilation` dans la classe `Program`. Vous y reviendrez pour remplir cette méthode :

Écrivez le code suivant pour effectuer une itération sur chaque SyntaxTree dans le test de Compilation. Pour chacune d'elles, initialisez un nouveau `TypeInferenceRewriter` avec le SemanticModel pour cet arborescence :

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Dans l'instruction `foreach` que vous avez créée, ajoutez le code suivant pour effectuer la transformation sur chaque arborescence source. Ce code écrit de façon conditionnelle la nouvelle arborescence transformée si des modifications ont été effectuées. Votre module de réécriture doit uniquement modifier une arborescence si elle rencontre une ou plusieurs déclarations de variables locales qui pourraient être simplifiées à l'aide de l'inférence de type :

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Vous devriez voir des tildes sous le code `File.WriteAllText`. Sélectionnez l'ampoule et ajoutez l'instruction `using System.IO;` nécessaire.

Vous avez presque terminé. Il y a une étape restante : création d'un test Compilation . Étant donné que vous n'avez pas utilisé du tout l'inférence de type pendant ce démarrage rapide, cela aurait été un cas de test parfait. Malheureusement, la création d'une compilation à partir d'un fichier de projet C# dépasse le cadre de cette procédure pas à pas. Heureusement, si vous avez suivi attentivement les instructions, il reste un espoir. Remplacez le contenu de la méthode `CreateTestCompilation` par le code suivant : Il crée une compilation de test correspondant par coïncidence au projet décrit dans ce démarrage rapide :

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                                .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                                .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
        MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
        MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Croisez les doigts et exécutez le projet. Dans Visual Studio, choisissez **Déboguer** > **Démarrer le débogage**. Visual Studio devrait vous avertir que les fichiers de votre projet ont été modifiés. Cliquez sur «**Oui pour tout**» pour recharger les fichiers modifiés. Examinez ces fichiers pour vérifier la qualité de votre travail. Remarquez à quel point le code serait plus propre sans tous ces spécificateurs de type explicites et redondants.

Félicitations ! Vous avez utilisé les **API du compilateur** pour écrire votre propre refactorisation qui recherche certains modèles syntaxiques dans tous les fichiers d'un projet C#, analyse la sémantique du code source correspondant à ces modèles puis la transforme. Vous êtes maintenant officiellement un auteur de refactorisation !

Tutoriel : Écrire votre premier analyseur et correctif de code

02/11/2020 • 46 minutes to read • Edit Online

Le SDK .NET Compiler Platform fournit les outils nécessaires pour créer des avertissements personnalisés qui ciblent C# ou le code Visual Basic. Votre **analyseur** contient le code qui reconnaît les violations de votre règle. Votre **correctif de code** contient le code qui résout la violation. Les règles que vous implémentez peuvent aller de la structure du code au style de codage et aux conventions d'affectation de noms, et bien plus encore. .NET Compiler Platform fournit le framework permettant d'exécuter l'analyse alors que les développeurs écrivent du code, et toutes les fonctionnalités de l'IU Visual Studio pour corriger le code : afficher des tildes dans l'éditeur, renseigner la liste d'erreurs Visual Studio, créer des suggestions « ampoule » et afficher un aperçu détaillé des corrections suggérées.

Dans ce tutoriel, vous allez explorer la création d'un **analyseur** et d'un **correctif de code** associé à l'aide des API Roslyn. Un analyseur consiste à effectuer une analyse du code source et signaler un problème à l'utilisateur. Un analyseur peut également fournir un correctif de code qui représente une modification du code source de l'utilisateur. Ce tutoriel crée un analyseur qui recherche des déclarations de variables locales qui pourraient être déclarées à l'aide du modificateur `const` mais qui ne le sont pas. Le correctif de code associé modifie ces déclarations pour ajouter le modificateur `const`.

Prérequis

- Visual Studio 2019 version 16,7 ou ultérieure

Vous devez installer le kit de **développement logiciel (SDK) .NET Compiler Platform** via le Visual Studio installer :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **l'éditeur DGML**

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**

4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Il existe plusieurs étapes pour créer et valider votre analyseur :

1. Créez la solution.
2. Enregistrez le nom de l'analyseur et sa description.
3. Créez un rapport sur les avertissements et les recommandations de l'analyseur.
4. Implémentez le correctif de code pour accepter les recommandations.
5. Améliorez l'analyse par le biais de tests unitaires.

Explorer le modèle d'analyseur

Votre analyseur signale à l'utilisateur les déclarations de variables locales qui peuvent être converties en constantes locales. Considérons par exemple le code suivant :

```
int x = 0;
Console.WriteLine(x);
```

Dans le code ci-dessus, une valeur constante est attribuée à `x` et n'est jamais modifiée. Elle peut être déclarée à l'aide du modificateur `const` :

```
const int x = 0;
Console.WriteLine(x);
```

L'analyse pour déterminer si une variable peut être déclarée constante est impliquée, nécessitant une analyse syntaxique, une analyse constante de l'expression de l'initialiseur et une analyse du flux de données pour s'assurer que la variable n'est jamais accessible en écriture. .NET Compiler Platform fournit les API qui simplifient l'exécution de cette analyse. La première étape consiste à créer un nouveau projet C# **Analyseur avec correctif de code**.

- Dans Visual Studio, choisissez **Fichier > Nouveau > Projet...** pour afficher la boîte de dialogue Nouveau projet.
- Sous **Visual C# > Extensibilité**, choisissez **Analyseur avec correctif de code (.NET Standard)**.
- Nommez votre projet « **MakeConst** », puis cliquez sur OK.

Le modèle Analyseur avec correctif de code crée trois projets : un contient l'analyseur et le correctif de code, le second est un projet de test unitaire et le troisième est le projet VSIX. Le projet de démarrage par défaut est le projet VSIX. Appuyez sur F5 pour démarrer le projet VSIX. Ceci démarre une deuxième instance de Visual Studio qui a chargé votre nouvel analyseur.

> **TIP**
> Lorsque vous exécutez votre analyseur, vous démarrez une deuxième copie de Visual Studio. Cette deuxième copie utilise un hive de Registre différent pour stocker les paramètres. Cela vous permet de différencier les paramètres Visual dans les deux copies de Visual Studio. Vous pouvez choisir un autre thème pour l'exécution expérimentale de Visual Studio. En outre, ne rendez pas vos paramètres itinérants et ne vous connectez pas à votre compte Visual Studio à l'aide de l'exécution expérimentale de Visual Studio. Cela permet de conserver les paramètres différents.

Dans la deuxième instance de Visual Studio que vous venez de démarrer, créez un projet d'application console C# (.NET Core ou .NET Framework projet fonctionne, les analyseurs fonctionnent au niveau de la source.) Placez le

curseur sur le jeton avec un soulignement ondulé et le texte d'avertissement fourni par un analyseur s'affiche.

Le modèle crée un analyseur qui émet un avertissement sur chaque déclaration de type dont le nom de type contient des lettres minuscules, comme indiqué dans la figure suivante :

```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Le modèle fournit également un correctif de code qui modifie n'importe quel nom de type contenant des caractères minuscules en majuscules. Vous pouvez cliquer sur l'ampoule affichée avec l'avertissement pour voir les modifications suggérées. Le fait d'accepter les modifications suggérées met à jour le nom de type et toutes les références à ce type dans la solution. Maintenant que vous avez vu l'analyseur initial en action, fermez la deuxième instance de Visual Studio et revenez à votre projet d'analyseur.

Vous n'êtes pas obligé de démarrer une deuxième copie de Visual Studio et de créer un nouveau code pour tester chaque modification de votre analyseur. Le modèle crée également un projet de test unitaire pour vous. Ce projet contient deux tests. `TestMethod1` montre le format classique d'un test qui analyse le code sans déclencher un diagnostic. `TestMethod2` montre le format d'un test qui déclenche un diagnostic, puis applique un correctif de code proposé. À mesure que vous générez votre analyseur et votre correctif de code, vous écrirez des tests pour des structures de codes différentes afin de vérifier votre travail. Les tests unitaires pour les analyseurs sont beaucoup plus rapides que de les tester de manière interactive avec Visual Studio.

> **TIP**
> Les tests unitaires d'analyseur sont un excellent outil lorsque vous savez quelles constructions de code doivent et ne doivent pas déclencher votre analyseur. Le chargement de votre analyseur dans une autre copie de Visual Studio est un excellent outil pour explorer et rechercher des constructions auxquelles vous n'avez peut-être pas encore pensé.

Créer des enregistrements d'analyseur

Le modèle crée la classe `DiagnosticAnalyzer` initiale, dans le fichier **MakeConstAnalyzer.cs**. Cet analyseur initial montre deux propriétés importantes de chaque analyseur.

- Chaque analyseur de diagnostic doit fournir un attribut `[DiagnosticAnalyzer]` qui décrit le langage sur lequel il opère.
- Chaque analyseur de diagnostic doit dériver de la classe DiagnosticAnalyzer.

Le modèle montre également les fonctionnalités de base qui font partie de n'importe quel analyseur :

1. Enregistrer les actions. Les actions représentent les modifications de code qui doivent déclencher votre analyseur pour qu'il examine les violations dans le code. Lorsque Visual Studio détecte des modifications de code qui correspondent à une action enregistré, il appelle la méthode enregistrée de votre analyseur.
2. Créer des diagnostics. Quand votre analyseur détecte une violation, il crée un objet de diagnostic que Visual Studio utilise pour informer l'utilisateur de la violation.

Vous enregistrez des actions dans votre substitution de la méthode DiagnosticAnalyzer.Initialize(AnalysisContext). Dans ce tutoriel, vous allez consulter des **nœuds de syntaxe** pour rechercher des déclarations locales et voir lesquelles ont des valeurs constantes. Si une déclaration peut être une constante, votre analyseur crée et signale un

diagnostic.

La première étape consiste à mettre à jour les constantes d'inscription et la méthode `Initialize` pour que ces constantes indiquent votre analyseur « Make Const ». La plupart des constantes de chaîne est définie dans le fichier de ressources de chaîne. Cette pratique facilite la localisation. Ouvrez le fichier **Resources.resx** pour le projet d'analyseur **MakeConst**. Cela affiche l'éditeur de ressources. Mettez à jour les ressources de chaîne comme suit :

- Remplacez `AnalyzerTitle` par « Variable can be made constant » (La variable peut être rendue constante).
- Remplacez `AnalyzerMessageFormat` par « Can be made constant » (Peut être rendu constant).
- Remplacez `AnalyzerDescription` par « Make constant » (Rendre constant).

Modifiez aussi le menu déroulant **Modificateur d'accès** sur `public`. Ainsi, ces constantes sont plus faciles à utiliser dans les tests unitaires. Lorsque vous avez terminé, l'éditeur de ressources doit apparaître comme le montre l'illustration suivante :

Name	Value	Comment
AnalyzerTitle	Variable can be made constant.	The title of the diagnostic.
AnalyzerMessageFormat	can be made constant	The format-able message the diagnostic displays.
AnalyzerDescription	Make Constant.	An optional longer localizable description of the diagnostic.

Les modifications restantes ont lieu dans le fichier de l'analyseur. Ouvrez **MakeConstAnalyzer.cs** dans Visual Studio. Modifiez l'action enregistrée d'une action qui agit sur les symboles à une action qui agit sur la syntaxe. Dans la méthode `MakeConstAnalyzerAnalyzer.Initialize`, recherchez la ligne qui enregistre l'action sur les symboles :

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Remplacez-la par la ligne suivante :

```
context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.Analyze | GeneratedCodeAnalysisFlags.None);
context.EnableConcurrentExecution();
context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.LocalDeclarationStatement);
```

Après cette modification, vous pouvez supprimer la méthode `AnalyzeSymbol`. Cet outil examine SyntaxKind.LocalDeclarationStatement, et non les instructions SymbolKind.NamedType. Notez que `AnalyzeNode` est souligné par des traits ondulés rouges. Le code que vous venez d'ajouter référence une méthode `AnalyzeNode` qui n'a pas été déclarée. Déclarez cette méthode en utilisant le code suivant :

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Remplacez `Category` par « Usage » (Utilisation) dans **MakeConstAnalyzer.cs** comme illustré dans le code suivant :

```
private const string Category = "Usage";
```

Rechercher des déclarations locales qui pourraient être constantes

Il est temps d'écrire la première version de la méthode `AnalyzeNode`. Elle doit rechercher une déclaration locale unique qui peut être `const` mais ne l'est pas, comme le code suivant :

```
int x = 0;
Console.WriteLine(x);
```

La première étape consiste à rechercher les déclarations locales. Ajoutez le code suivant à `AnalyzeNode` dans **MakeConstAnalyzer.cs** :

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Ce cast réussit toujours, car votre analyseur a enregistré les modifications apportées aux déclarations locales, et uniquement les déclarations locales. Aucun autre type de nœud ne déclenche un appel à votre méthode `AnalyzeNode`. Ensuite, vérifiez la présence de modificateurs `const` dans la déclaration. Si vous en trouvez, retournez immédiatement. Le code suivant recherche les modificateurs `const` sur la déclaration locale :

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Enfin, vous devez vérifier que la variable peut être `const`. Cela signifie de s'assurer qu'elle n'est jamais assignée après son initialisation.

Vous allez effectuer une analyse sémantique à l'aide de SyntaxNodeAnalysisContext. Vous utilisez l'argument `context` pour déterminer si la déclaration de variable locale peut être rendue `const`. Un Microsoft.CodeAnalysis.SemanticModel représente toutes les informations sémantiques dans un fichier source unique. Vous pouvez en apprendre plus dans l'article qui traite des modèles sémantiques. Vous allez utiliser Microsoft.CodeAnalysis.SemanticModel pour effectuer une analyse de flux de données sur l'instruction de déclaration locale. Ensuite, vous utilisez les résultats de cette analyse de flux de données pour vous assurer que la variable locale n'est pas écrite avec une nouvelle valeur ailleurs. Appelez la méthode d'extension GetDeclaredSymbol pour récupérer ILocalSymbol pour la variable et vérifiez qu'il n'est pas contenu avec la collection DataFlowAnalysis.WrittenOutside de l'analyse de flux de données. Ajoutez le code suivant à la fin de la méthode `AnalyzeNode` :

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

Le code qui vient d'être ajouté garantit que la variable n'est pas modifiée et peut par conséquent devenir `const`. Il est temps de générer le diagnostic. Ajoutez le code suivant comme dernière ligne dans `AnalyzeNode` :

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation()));
```

Vous pouvez vérifier votre progression en appuyant sur F5 pour exécuter votre analyseur. Vous pouvez charger l'application console que vous avez créée précédemment et ajouter le code de test suivant :

```
int x = 0;
Console.WriteLine(x);
```

L'ampoule doit apparaître et votre analyseur doit signaler un diagnostic. Toutefois, l'ampoule utilise toujours le correctif de code généré par le modèle et vous indique qu'il peut être converti en majuscules. La section suivante explique comment écrire le correctif de code.

Écrire le correctif de code

Un analyseur peut fournir un ou plusieurs correctifs de code. Un correctif de code définit une modification qui traite le problème signalé. Pour l'analyseur que vous avez créé, vous pouvez fournir un correctif de code qui insère le mot clé const :

```
const int x = 0;
Console.WriteLine(x);
```

L'utilisateur le choisit à partir de l'IU ampoule dans l'éditeur et Visual Studio modifie le code.

Ouvrez le fichier **MakeConstCodeFixProvider.cs** ajouté par le modèle. Ce correctif de code est déjà associé à l'ID de diagnostic produit par votre analyseur de diagnostic, mais il n'implémente pas encore la transformation de code adéquate. Tout d'abord, vous devez supprimer une partie du code du modèle. Remplacez la chaîne de titre par « Make constant » :

```
private const string title = "Make constant";
```

Ensuite, supprimez la méthode `MakeUppercaseAsync`. Elle n'est plus applicable.

Tous les fournisseurs de correctif de code dérivent de CodeFixProvider. Ils substituent tous CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) pour signaler les correctifs de code disponibles. Dans `RegisterCodeFixesAsync`, remplacez le type de nœud ancêtre que vous recherchez par LocalDeclarationStatementSyntax pour faire correspondre le diagnostic :

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>
().First();
```

Ensuite, modifiez la dernière ligne pour enregistrer un correctif de code. Votre correctif crée un nouveau document obtenu en ajoutant le modificateur `const` à une déclaration existante :

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: title,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: title),
    diagnostic);
```

Notez les traits de soulignement ondulés rouges dans le code que vous venez d'ajouter sur le symbole `MakeConstAsync`. Ajoutez une déclaration pour `MakeConstAsync` comme le code suivant :

```
private async Task<Document> MakeConstAsync(Document document,
   LocalDeclarationStatementSyntax localDeclaration,
   CancellationToken cancellationToken)
{
}
```

Votre nouvelle méthode `MakeConstAsync` transformera Document qui représente le fichier source de l'utilisateur en un nouveau Document qui contient à présent une déclaration `const`.

Vous créez un nouveau jeton de mot clé `const` à insérer au début de l'instruction de déclaration. Veillez tout d'abord à supprimer les trivia de début dans le premier jeton de l'instruction de déclaration et à les associer au jeton `const`. Ajoutez le code suivant à la méthode `MakeConstAsync` :

```
// Remove the leading trivia from the local declaration.
var firstToken = localDeclaration.GetFirstToken();
var leadingTrivia = firstToken.LeadingTrivia;
var trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Ensuite, ajoutez le jeton `const` dans la déclaration à l'aide du code suivant :

```
// Insert the const token into the modifiers list, creating a new modifiers list.
var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
var newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Ensuite, mettez en forme la nouvelle déclaration en fonction des règles de mise en forme en C#. La mise en forme de vos modifications pour correspondre au code existant crée une meilleure expérience. Ajoutez l'instruction suivante immédiatement après le code existant :

```
// Add an annotation to format the new local declaration.
var formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Un nouvel espace de noms est requis pour ce code. Ajoutez la directive `using` suivante en haut du fichier :

```
using Microsoft.CodeAnalysis.Formatting;
```

L'étape finale consiste à apporter votre modification. Ce processus comprend trois étapes :

1. Obtenir un handle pour le document existant.
2. Créer un document en remplaçant la déclaration existante par la nouvelle déclaration.
3. Retourner le nouveau document.

Ajoutez le code suivant à la fin de la méthode `MakeConstAsync` :

```
// Replace the old local declaration with the new local declaration.
var oldRoot = await document.GetSyntaxRootAsync(cancellationToken);
var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Votre correctif de code est prêt à être testé. Appuyez sur F5 pour exécuter le projet de l'analyseur dans une seconde instance de Visual Studio. Dans la deuxième instance de Visual Studio, créez un projet Application console C# et ajoutez quelques déclarations de variables locales initialisées avec des valeurs constantes à la méthode Main. Vous verrez que ces éléments sont signalés en tant qu'avertissements comme indiqué ci-dessous.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

Vous avez bien progressé. Des traits de soulignement ondulés s'affichent sous les déclarations qui peuvent devenir `const`. Mais il reste encore du travail. Cela fonctionne si vous ajoutez `const` aux déclarations qui commencent par `i`, puis `j` et enfin `k`. Mais, si le modificateur `const` est ajouté dans un autre ordre, en commençant par `k`, l'analyseur crée des erreurs : `k` ne peut pas être déclaré `const`, sauf si `i` et `j` sont tous deux déjà `const`. Vous devez effectuer d'autres analyses afin de vous assurer que vous gérez les différentes façons dont les variables peuvent être déclarées et initialisées.

Générer des tests pilotés par les données

Votre analyseur et votre correctif de code travaillent sur un cas simple d'une déclaration unique qui peut être devenir constante. Il existe de nombreuses instructions de déclaration possibles où cette implémentation commet des erreurs. Vous allez traiter ces cas en travaillant avec la bibliothèque de tests unitaires écrite par le modèle. C'est beaucoup plus rapide que d'ouvrir plusieurs fois une deuxième copie de Visual Studio.

Ouvrez le fichier **MakeConstUnitTests.cs** dans le projet de test unitaire. Le modèle a créé deux tests qui suivent les deux modèles courants pour un test unitaire d'analyseur et de correctif de code. `TestMethod1` montre le modèle pour un test qui garantit que l'analyseur ne signale pas un diagnostic lorsqu'il ne le devrait pas. `TestMethod2` montre le modèle pour créer un rapport de diagnostic et exécuter le correctif de code.

Le code de presque chaque test pour votre analyseur suit un de ces deux modèles. Pour la première étape, vous pouvez revoir ces tests en tant que tests pilotés par les données. Ensuite, il sera facile de créer de nouveaux tests en ajoutant de nouvelles constantes de chaîne pour représenter les différentes entrées de test.

Pour plus d'efficacité, la première étape consiste à refactoriser les deux tests en tests pilotés par des données. Ensuite, il vous suffit de définir quelques constantes de chaîne pour chaque nouveau test. Pendant la refactorisation, renommez les deux méthodes avec des noms plus performants. Remplacez `TestMethod1` par ce test qui garantit qu'aucun diagnostic n'est déclenché :

```
[DataTestMethod]
[DataRow("")]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}
```

Vous pouvez créer une nouvelle ligne de données pour ce test en définissant un fragment de code qui ne doit pas entraîner votre diagnostic à déclencher un avertissement. Cette surcharge de passes `VerifyCSharpDiagnostic`

lorsqu'il n'y a aucun diagnostic déclenché pour le fragment de code source.

Ensuite, remplacez `TestMethod2` par ce test qui garantit qu'un diagnostic est déclenché et un correctif de code appliqué pour le fragment de code source :

```
[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13)]
public void WhenDiagnosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
            new[] {
                    new DiagnosticResultLocation("Test0.cs", line, column)
                }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}
```

Le code précédent a apporté également quelques modifications au code qui génère le résultat de diagnostic attendu. Il utilise les constantes publiques enregistrées dans l'analyseur `MakeConst`. En outre, il utilise deux constantes de chaîne pour la source d'entrée et corrigée. Ajoutez les constantes de chaîne suivantes à la classe `UnitTest` :

```
private const string LocalIntCouldBeConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i);
        }
    }
}";

private const string LocalIntCouldBeConstantFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

Exécutez ces deux tests pour vous assurer qu'ils réussissent. Dans Visual Studio, ouvrez **l'Explorateur de tests** en sélectionnant **Test** > **Windows** > **Explorateur de tests**. Sélectionnez ensuite le lien **exécuter tout** .

Créer des tests pour les déclarations valides

En règle générale, les analyseurs doivent s'arrêter aussi rapidement que possible, en effectuant un minimum de travail. Visual Studio appelle les analyseurs enregistrés lorsque l'utilisateur modifie le code. La réactivité est essentielle. Il existe plusieurs cas de test pour le code qui ne doivent pas déclencher votre diagnostic. Votre analyseur a déjà géré l'un de ces tests, le cas où une variable est assignée après avoir été initialisée. Ajoutez la constante de chaîne suivante à vos tests pour représenter ce cas :

```
private const string VariableAssigned = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i++);
        }
    }
}";
```

Ensuite, ajoutez une ligne de données pour ce test, comme indiqué dans l'extrait de code ci-dessous :

```
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Ce test réussit également. Ensuite, ajoutez des constantes pour les conditions que vous n'avez pas encore gérées :

- Déclarations qui sont déjà `const`, car elles sont déjà constantes :

```
private const string AlreadyConst = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

- Déclarations sans initialiseur, car il n'existe aucune valeur à utiliser :

```
private const string NoInitializer = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;
            i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

- Déclarations où l'initialiseur n'est pas une constante, car elles ne peuvent pas être des constantes de compilation :

```
private const string InitializerNotConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = DateTime.Now.DayOfYear;
            Console.WriteLine(i);
        }
    }
}";
```

Cela peut être encore plus compliqué, car C# accepte que plusieurs déclarations forment une seule instruction. Prenez en compte la constante de chaîne de cas de test suivante :

```
private const string MultipleInitializers = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0, j = DateTime.Now.DayOfYear;
            Console.WriteLine(i, j);
        }
    }
}";
```

La variable `i` peut devenir constante, mais la variable `j` ne le peut pas. Par conséquent, cette instruction ne peut pas devenir une déclaration constante. Ajouter les déclarations `DataRow` pour tous ces tests :

```
[DataTestMethod]
[DataRow(""),
    DataRow(VariableAssigned),
    DataRow(AlreadyConst),
    DataRow(NoInitializer),
    DataRow(InitializerNotConstant),
    DataRow(MultipleInitializers)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Réexécutez vos tests, et vous verrez ces nouveaux cas de test échouent.

Mettre à jour votre analyseur afin d'ignorer les déclarations correctes

Vous avez besoin de certaines améliorations dans la méthode `AnalyzeNode` de votre analyseur pour éliminer le code correspondant à ces conditions. Ce sont toutes des conditions associées, donc des modifications similaires résoudront toutes ces conditions. Dans `AnalyzeNode`, effectuez les changements suivants :

- Votre analyse sémantique a examiné une déclaration de variable unique. Ce code doit se trouver dans une boucle `foreach` qui examine toutes les variables déclarées dans la même instruction.
- Chaque variable déclarée doit avoir un initialiseur.
- L'initialiseur de chaque variable déclarée doit être une constante de compilation.

Dans votre méthode `AnalyzeNode`, remplacez l'analyse sémantique d'origine :

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

par l'extrait de code suivant :

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (var variable in localDeclaration.Declaration.Variables)
{
    var initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    var constantValue = context.SemanticModel.GetConstantValue(initializer.Value);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (var variable in localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

La première boucle `foreach` examine chaque déclaration de variable à l'aide de l'analyse syntaxique. La première vérification garantit que la variable a un initialiseur. La deuxième vérification garantit que l'initialiseur est une constante. La deuxième boucle contient l'analyse sémantique d'origine. Les vérifications sémantiques sont dans une boucle distincte, car elles ont un impact plus important sur les performances. Réexécutez vos tests. Ils doivent tous réussir.

Ajouter la touche finale

Vous avez presque terminé. Il existe quelques conditions de plus que votre analyseur doit gérer. Visual Studio appelle les analyseurs lorsque l'utilisateur écrit du code. Souvent, votre analyseur sera appelé lorsque du code ne se compile pas. La méthode `AnalyzeNode` de votre analyseur de diagnostic ne vérifie pas si la valeur de constante est convertible en type de variable. Par conséquent, l'implémentation actuelle convertit sans problème une déclaration incorrecte comme int i = « abc » en une constante locale. Ajoutez une constante de chaîne source pour cette condition :

```
private const string DeclarationIsInvalid = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = ""abc"";
        }
    }
}";
```

En outre, les types de référence ne sont pas gérés correctement. La seule valeur de constante autorisée pour un

type de référence est `null`, sauf dans le cas de System.String, qui autorise des littéraux de chaîne. En d'autres termes, `const string s = "abc"` est légal, mais pas `const object s = "abc"`. Cet extrait de code vérifie cette condition :

```
private const string ReferenceTypeIsntString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            object s = ""abc"";
        }
    }
}";
```

Pour être exhaustif, vous devez ajouter un autre test afin de vous assurer que vous pouvez créer une déclaration de constante pour une chaîne. L'extrait de code suivant définit le code qui déclenche le diagnostic et le code une fois que le correctif a été appliqué :

```
private const string ConstantIsString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = ""abc"";
        }
    }
}";

private const string ConstantIsStringFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const string s = ""abc"";
        }
    }
}";
```

Enfin, si une variable est déclarée avec le mot clé `var`, le correctif de code fait la mauvaise chose et génère une déclaration `const var`, qui n'est pas prise en charge par le langage C#. Pour corriger ce bogue, le correctif de code doit remplacer le mot clé `var` par le nom du type déduit :

```
private const string DeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = 4;
        }
    }
}";

private const string DeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int item = 4;
        }
    }
}";
private const string StringDeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = ""abc"";
        }
    }
}";
private const string StringDeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const string item = ""abc"";
        }
    }
}";
```

Ces modifications mettent à jour les déclarations de ligne de données pour les deux tests. Le code suivant montre ces tests avec tous les attributs de ligne de données :

```
//No diagnostics expected to show up
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned),
 DataRow(AlreadyConst),
 DataRow(NoInitializer),
 DataRow(InitializerNotConstant),
 DataRow(MultipleInitializers),
 DataRow(DeclarationIsInvalid),
 DataRow(ReferenceTypeIsntString)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}

[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13),
 DataRow(ConstantIsString, ConstantIsStringFixed, 10, 13),
 DataRow(DeclarationUsesVar, DeclarationUsesVarFixedHasType, 10, 13),
 DataRow(StringDeclarationUsesVar, StringDeclarationUsesVarFixedHasType, 10, 13)]
public void WhenDiagosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
            new[] {
                    new DiagnosticResultLocation("Test0.cs", line, column)
                }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}
```

Heureusement, tous les bogues ci-dessus peuvent être gérés à l'aide des mêmes techniques que celles que vous venez d'apprendre.

Pour corriger le premier bogue, commencez par ouvrir **DiagnosticAnalyzer.cs** et recherchez la boucle foreach où chacun des initialiseurs de la déclaration locale est vérifié pour s'assurer que des valeurs constantes leur sont attribués. Immédiatement *avant* la première boucle foreach, appelez `context.SemanticModel.GetTypeInfo()` pour récupérer des informations détaillées sur le type déclaré de la déclaration locale :

```
var variableTypeName = localDeclaration.Declaration.Type;
var variableType = context.SemanticModel.GetTypeInfo(variableTypeName).ConvertedType;
```

Ensuite, dans votre boucle `foreach`, vérifiez chaque initialiseur pour vous assurer qu'il peut être converti dans le type de variable. Ajoutez la vérification suivante après avoir vérifié que l'initialiseur est une constante :

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
var conversion = context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

La modification suivante s'appuie sur la précédente. Avant l'accolade fermante de la première boucle foreach, ajoutez le code suivant pour vérifier le type de la déclaration locale si la constante est une chaîne ou une valeur null.

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Vous devez écrire un peu plus de code dans votre fournisseur de correctifs de code pour remplacer le `var` mot clé par le nom de type correct. Revenez dans **CodeFixProvider.cs**. Le code que vous ajouterez effectue les étapes suivantes :

- Vérifiez si la déclaration est une déclaration `var`, et dans ce cas :
- Créez un nouveau type pour le type déduit.
- Assurez-vous que la déclaration de type n'est pas un alias. Le cas échéant, il est permis de déclarer `const var`.
- Assurez-vous que `var` n'est pas un nom de type dans ce programme. (Dans ce cas, `const var` est autorisé).
- Simplifier le nom de type complet

On dirait que beaucoup de code est nécessaire pour cela. Ce n'est pas le cas. Remplacez la ligne qui déclare et initialise `newLocal` par le code suivant. Il se place immédiatement après l'initialisation de `newModifiers` :

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
var variableDeclaration = localDeclaration.Declaration;
var variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    var semanticModel = await document.GetSemanticModelAsync(cancellationToken);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        var type = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            var typeName = SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            var simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
var newLocal = trimmedLocal.WithModifiers(newModifiers)
                           .WithDeclaration(variableDeclaration);
```

Vous devez ajouter une `using` directive pour utiliser le Simplifier type :

```
using Microsoft.CodeAnalysis.Simplification;
```

Exécutez vos tests. Ils doivent tous réussir. Félicitez-vous en exécutant votre analyseur terminé. Appuyez sur `CTRL + F5` pour exécuter le projet de l'analyseur dans une seconde instance de Visual Studio avec l'extension de prévisualisation Roslyn chargée.

- Dans la deuxième instance de Visual Studio, créez un projet Application console C# et ajoutez `int x = "abc";` à la méthode Main. Grâce à la correction du premier bogue, aucun avertissement ne doit être signalé pour cette déclaration de variable locale (même s'il existe une erreur du compilateur comme prévu).
- Ensuite, ajoutez `object s = "abc";` à la méthode Main. En raison de la correction du deuxième bogue, aucun avertissement ne doit être signalé.
- Enfin, ajoutez une autre variable locale qui utilise le mot clé `var`. Un avertissement est signalé et une suggestion apparaît en dessous à gauche.
- Déplacez le signe insertion de l'éditeur sur le soulignement ondulé et appuyez sur `CTRL + .`. pour afficher le correctif de code suggéré. Après avoir sélectionné votre correctif de code, Notez que le `var` mot clé est désormais géré correctement.

Enfin, ajoutez le code suivant :

```
int i = 2;
int j = 32;
int k = i + j;
```

Après ces modifications, vous obtenez des traits de soulignement ondulés rouges uniquement sur les deux premières variables. Ajoutez `const` à `i` et à `j`, et vous obtenez un nouvel avertissement sur `k`, car cet élément peut désormais devenir `const`.

Félicitations ! Vous avez créé votre première extension .NET Compiler Platform qui effectue une analyse de code à la volée pour détecter un problème et fournit une solution rapide pour résoudre ce problème. Tout au long du processus, vous avez découvert la plupart des API de code qui font partie du SDK .NET Compiler Platform (API Roslyn). Vous pouvez comparer votre travail avec l'exemple terminé dans notre référentiel GitHub d'exemples.

Autres ressources

- Bien démarrer avec l'analyse de la syntaxe
- Bien démarrer avec l'analyse sémantique

Guide de programmation C#

02/11/2020 • 2 minutes to read • Edit Online

Cette section fournit des informations détaillées sur les fonctionnalités clés du langage C# et les fonctionnalités accessibles à C# via .NET.

La majeure partie de cette section suppose que vous connaissez déjà en partie C# et les concepts de programmation généraux. Si vous êtes débutant en programmation ou en C#, vous souhaiterez peut-être consulter la Présentation des didacticiels C# ou du didacticiel .net dans le navigateur, où aucune connaissance préalable de la programmation n'est requise.

Pour plus d'informations sur les mots clés, les opérateurs et les directives de préprocesseur spécifiques, consultez référence C#. Pour plus d'informations sur la spécification du langage C#, consultez la page Spécification du langage C#.

Sections du programme

À l'intérieur d'un programme C#

Main () et arguments de ligne de commande

Sections du langage

Instructions, expressions et opérateurs

Types

Classes et structs

Interfaces

Délégués

Tableaux

Chaînes

Propriétés

Indexeurs

Événements

Génériques

Iterators

Expressions de requête LINQ

Espaces de noms

Pointeurs et code unsafe

Commentaires de documentation XML

Sections de la plateforme

Domaines d'application

Assemblys dans .NET

Attributs

Regroupements

Exceptions et gestion des exceptions

Système de fichiers et Registre (Guide de programmation C#)

Interopérabilité

Réflexion

Voir aussi

- Référence C#

À l'intérieur d'un programme C#

02/11/2020 • 2 minutes to read • Edit Online

La section traite de la structure générale d'un programme C# et inclut l'exemple standard « Hello, World! ».

Contenu de cette section

- Hello World -- Votre premier programme
- Structure générale d'un programme C#
- Noms d'identificateurs
- Conventions de codage C#

Sections connexes

- Bien démarrer avec C#
- Guide de programmation C#
- Référence C#
- Exemples et tutoriels

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#

Hello World--votre premier programme

02/11/2020 • 8 minutes to read • Edit Online

Dans cet article, vous allez utiliser Visual Studio pour créer le « Hello World ! » traditionnel . Visual Studio est un environnement de développement intégré (IDE) professionnel avec de nombreuses fonctionnalités conçues pour le développement .NET. Vous n'utiliserez que quelques-unes des fonctionnalités de Visual Studio pour créer ce programme. Pour en savoir plus sur Visual Studio, consultez prise en main avec Visual C#.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Créer une application

- Windows
- macOS

Démarrez Visual Studio. L'image suivante s'affiche sur Windows :

Sélectionnez **créer un nouveau projet** dans le coin inférieur droit de l'image. Visual Studio affiche la boîte **de dialogue Nouveau projet** :

> **NOTE**
>
> S'il s'agit de la première fois que vous démarrez Visual Studio, la liste des **modèles de projet récents** est vide.

Dans la boîte de dialogue Nouveau projet, choisissez application console (.NET Core), puis cliquez sur **suivant**. Donnez un nom à votre projet, tel que « HelloWorld », puis appuyez sur **créer**.

Visual Studio ouvre votre projet. Il s'agit déjà d'un « Hello World ! » de base. « Hello, World! ». Appuyez sur `Ctrl` + `F5` pour exécuter votre projet. Visual Studio génère votre projet et convertit le code source en un fichier exécutable. Ensuite, il lance une fenêtre de commande qui exécute votre nouvelle application. Vous devez voir le texte suivant dans la fenêtre :

```
Hello World!

C:\Program Files\dotnet\dotnet.exe (process 11964) exited with code 0.
Press any key to close this window . . .
```

Appuyez sur une touche pour fermer la fenêtre.

Éléments d'un programme C#

Examinons les parties importantes de ce programme. La première ligne contient un commentaire. Les caractères `//` convertissent le reste de la ligne en un commentaire.

```
// A Hello World! program in C#.
```

Vous pouvez également créer un commentaire à partir d'un bloc de texte en le plaçant entre les caractères `/*` et `*/`. Cela est illustré par l'exemple suivant.

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

Une application console C# doit contenir une méthode `Main`, dans laquelle le contrôle commence et termine. C'est dans la méthode `Main` que vous créez des objets et exécutez d'autres méthodes.

La méthode `Main` est une méthode static qui réside dans une classe ou un struct. Dans l'exemple « Hello World ! » précédent, il réside dans une classe nommée `Hello`. Vous pouvez déclarer la méthode `Main` de l'une des façons suivantes :

- Elle peut retourner `void`. Cela signifie que votre programme ne retourne pas de valeur.

```
static void Main()
{
    //...
}
```

- Elle peut également retourner un entier. L'entier est le **Code de sortie** de votre application.

```
static int Main()
{
    //...
    return 0;
}
```

- Elle peut accepter des arguments avec n'importe quel type de retour.

```
static void Main(string[] args)
{
    //...
}
```

-ou-

```
static int Main(string[] args)
{
    //...
    return 0;
}
```

Le paramètre `args` de la méthode `Main` est un tableau `string` qui contient les arguments de la ligne de commande utilisés pour appeler le programme.

Pour plus d'informations sur l'utilisation des arguments de ligne de commande, consultez les exemples dans main () et arguments de ligne de commande.

Entrée et sortie

Les programmes C# utilisent généralement les services d'entrée/sortie fournis par la bibliothèque Runtime de .NET. L'instruction `System.Console.WriteLine("Hello World!");` utilise la méthode WriteLine. C'est l'une des méthodes de sortie de la classe Console de la bibliothèque runtime. Elle affiche son paramètre de chaîne dans le flux de sortie standard suivi d'une nouvelle ligne. D'autres méthodes Console sont disponibles pour différentes opérations d'entrée et de sortie. Si vous incluez la directive `using System;` au début du programme, vous pouvez utiliser directement les classes et les méthodes System sans les qualifier complètement. Par exemple, vous pouvez appeler

`Console.WriteLine` au lieu de `System.Console.WriteLine` :

```
using System;
```

```
Console.WriteLine("Hello World!");
```

Pour plus d'informations sur les méthodes d'entrée/sortie, consultez System.IO.

Voir aussi

- Guide de programmation C#
- Exemples et tutoriels
- Main () et arguments de ligne de commande
- Mise en route de Visual C#

Structure générale d'un programme C# (guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Les programmes C# se composent d'un ou plusieurs fichiers. Chaque fichier peut contenir zéro ou plusieurs espaces de noms. Un espace de noms peut contenir des types tels que des classes, des structs, des interfaces, des énumérations et des délégués, en plus d'autres espaces de noms. Voici la structure d'un programme C# qui contient tous ces éléments.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

Sections connexes

Pour plus d'informations :

- Classes
- Structures
- Espaces de noms

- Interfaces
- Délégués

Spécification du langage C#

Pour plus d'informations, consultez Concepts de base dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- À l'intérieur d'un programme C#
- Référence C#

Noms d'identificateurs

18/03/2020 • 2 minutes to read • Edit Online

Un **identificateur** est le nom que vous assignez à un type (classe, interface, struct, délégué ou enum), un membre, une variable ou un espace de noms. Les identificateurs valides doivent suivre les règles suivantes :

- Les identificateurs doivent commencer par une lettre ou par `_`.
- Les identificateurs peuvent contenir des lettres Unicode, des nombres décimaux, des caractères de connexion Unicode, des caractères de combinaison Unicode ou des caractères de mise en forme Unicode. Pour plus d'informations sur les catégories Unicode, consultez la Base de données des catégories Unicode. Vous pouvez déclarer des identificateurs qui correspondent à des mots clés C# en utilisant le préfixe `@` sur l'identificateur. `@` ne fait pas partie du nom de l'identificateur. Par exemple, `@if` déclare un identificateur nommé `if`. Ces identificateurs textuels sont destinés principalement à assurer l'interopérabilité avec les identificateurs déclarés dans d'autres langages.

Pour une définition complète des identificateurs valides, consultez la rubrique Identificateurs dans la spécification du langage C#.

Conventions d'affectation de noms

En plus des règles, un certain nombre de conventions de nommage des identificateurs sont utilisées dans les API .NET. Par convention, les programmes C# utilisent `PascalCase` pour les noms de types, les espaces de noms et tous les membres publics. De plus, les conventions suivantes sont communément respectées :

- Les noms d'interfaces commencent par un `I` majuscule.
- Les types d'attribut finissent par le mot `Attribute`.
- Les types enum utilisent un nom singulier pour les non-indicateurs et un nom pluriel pour les indicateurs.
- Les identificateurs ne doivent pas contenir deux caractères `_` consécutifs. Ces noms sont réservés aux identificateurs générés par le compilateur.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- À l'intérieur d'un programme C#
- Référence C
- Classes
- Types de structure
- Espaces de noms
- Interfaces
- Délégués

Conventions de codage C# (Guide de programmation C#)

02/11/2020 • 14 minutes to read • Edit Online

Les conventions de codage répondent aux objectifs suivants :

- Elles confèrent une apparence cohérente au code, afin que les lecteurs puissent se concentrer sur le contenu, pas sur la disposition.
- Elles permettent aux lecteurs de comprendre le code plus rapidement en formulant des hypothèses selon leur expérience antérieure.
- Elles facilitent la copie, la modification et la gestion du code.
- Elles illustrent les bonnes pratiques en C#.

Les instructions de cet article sont utilisées par Microsoft pour développer des exemples et de la documentation.

Conventions d'affectation de noms

- Dans les exemples courts qui ne comprennent pas de directives, utilisez les qualifications d'espace de noms. Si vous savez qu'un espace de noms est importé par défaut dans un projet, vous n'êtes pas obligé de qualifier entièrement les noms à partir de cet espace de noms. Les noms qualifiés peuvent être interrompus après un point (.) s'ils sont trop longs pour contenir sur une ligne unique, comme indiqué dans l'exemple suivant.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- Vous n'avez pas à modifier les noms des objets qui ont été créés à l'aide des outils du concepteur Visual Studio pour les rendre conformes aux autres directives.

Conventions de disposition

Une bonne disposition utilise la mise en forme pour souligner la structure de votre code et en faciliter la lecture. Les exemples Microsoft et autres se conforment aux conventions suivantes :

- Utilisez les paramètres par défaut de l'éditeur de code (retrait intelligent, retrait de quatre caractères, tabulations enregistrées en tant qu'espaces). Pour plus d'informations, consultez Options, Éditeur de texte, C#, Mise en forme.
- Écrivez une seule instruction par ligne.
- Écrivez une seule déclaration par ligne.
- Si les lignes de continuation ne sont pas mises en retrait automatiquement, indentez-les à l'aide d'un taquet de tabulation (quatre espaces).
- Ajoutez au moins une ligne blanche entre les définitions des méthodes et les définitions des propriétés.
- Utilisez des parenthèses pour rendre apparentes les clauses d'une expression, comme illustré dans le code suivant.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Conventions de commentaires

- Placez le commentaire sur une ligne séparée, pas à la fin d'une ligne de code.
- Commencez le commentaire par une lettre majuscule.
- Terminez le texte de commentaire par un point.
- Insérez un espace entre le délimiteur de commentaire (//) et le texte du commentaire, comme illustré dans l'exemple suivant.

```
// The following declaration creates a query. It does not run
// the query.
```

- Ne créez pas de blocs d'astérisques mis en forme autour des commentaires.

Directives du langage

Les sections suivantes décrivent les pratiques que l'équipe C# suit pour préparer les exemples de code et autres.

String, type de données

- Utilisez une interpolation de chaîne pour concaténer les chaînes courtes, comme illustré dans le code suivant.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- Pour ajouter des chaînes dans des boucles, en particulier lorsque vous travaillez avec une quantité de texte importante, utilisez un objet StringBuilder.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Variables locales implicitement typées

- Utilisez le typage implicite pour les variables locales quand le type de la variable est évident à droite de l'assignation ou quand le type précis n'importe pas.

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
```

- N'utilisez pas var quand le type n'est pas apparent à droite de l'assignation.

```
// When the type of a variable is not clear from the context, use an
// explicit type. You generally don't assume the type clear from a method name.
// A variable type is considered clear if it's a new operator or an explicit cast.
int var3 = Convert.ToInt32(Console.ReadLine());
int var4 = ExampleClass.ResultSoFar();
```

- Ne vous basez pas sur le nom de la variable pour spécifier son type. Il peut ne pas être correct.

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Évitez d'utiliser `var` à la place de dynamic.
- Utilisez le typage implicite pour déterminer le type de la variable de boucle dans les boucles for .

  L'exemple suivant utilise un typage implicite dans une instruction `for`.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- N'utilisez pas de typage implicite pour déterminer le type de la variable de boucle dans les boucles foreach .

  L'exemple suivant utilise un typage explicite dans une `foreach` instruction.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

> **NOTE**
>
> Veillez à ne pas modifier accidentellement un type d'élément de la collection Iterable. Par exemple, il est facile de passer de System.Linq.IQueryable à System.Collections.IEnumerable dans une `foreach` instruction, ce qui modifie l'exécution d'une requête.

Type de données non signé

En règle générale, utilisez `int` plutôt que les types non signés. L'utilisation de `int` est commun en C# et il est plus facile d'interagir avec d'autres bibliothèques lorsque vous utilisez `int`.

Tableaux

Utilisez la syntaxe concise lorsque vous initialisez des tableaux sur la ligne de déclaration.

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Délégués

Utilisez la syntaxe concise pour créer des instances d'un type délégué.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

Instructions try-catch et using dans la gestion des exceptions

- Utilisez une instruction try-catch pour la plus grande part de la gestion des exceptions.

  ```
  static string GetValueFromArray(string[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (System.IndexOutOfRangeException ex)
      {
          Console.WriteLine("Index is out of range: {0}", index);
          throw;
      }
  }
  ```

- Simplifiez votre code à l'aide de l'instruction using C#. Si vous avez une instruction try-finally dans laquelle le seul code du bloc `finally` est un appel à la méthode Dispose, utilisez à la place une instruction `using`.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

Opérateurs && et ||

Pour éviter les exceptions et accroître les performances en ignorant les comparaisons inutiles, utilisez à `&&` la place de `&` et de `||` au lieu de `|` quand vous effectuez des comparaisons, comme illustré dans l'exemple suivant.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

New, opérateur

- Utilisez la forme concise d'instanciation d'objets, avec un typage implicite, comme indiqué dans la déclaration suivante.

  ```
  var instance1 = new ExampleClass();
  ```

  La ligne précédente est équivalente à la déclaration suivante.

  ```
  ExampleClass instance2 = new ExampleClass();
  ```

- Utilisez les initialiseurs d'objets pour simplifier la création d'objet.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Gestion des événements

Si vous définissez un gestionnaire d'événements que vous n'avez pas besoin de supprimer ultérieurement, utilisez une expression lambda.

```
public Form2()
{
    // You can use a lambda expression to define an event handler.
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

```
// Using a lambda expression shortens the following traditional definition.
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Membres static

Appelez les membres static en utilisant le nom de la classe : *Nom_classe.Membre_statique*. Cette pratique rend le code plus lisible en clarifiant l'accès aux membres static. Ne qualifiez pas un membre static défini dans une classe de base avec le nom d'une classe dérivée. Lorsque ce code est compilé, la lisibilité du code est trompeuse et le code peut s'interrompre à l'avenir si vous ajoutez à la classe dérivée un membre static de même nom.

Requêtes LINQ

- Utilisez des noms explicites pour les variables de requête. L'exemple suivant utilise `seattleCustomers` pour les clients qui se trouvent à Seattle.

  ```
  var seattleCustomers = from customer in customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- Utilisez des alias pour vous assurer que les noms de propriétés des types anonymes sont correctement écrits en majuscules à l'aide de la casse Pascal.

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Renommez les propriétés lorsque les noms de propriétés dans le résultat sont ambigus. Par exemple, si votre requête retourne un nom de client et un ID de distributeur, au lieu de les laisser sous la forme `Name` et `ID` dans le résultat, renommez-les pour montrer clairement que `Name` est le nom d'un client et `ID` l'ID d'un distributeur.

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };
```

- Utilisez le typage implicite dans la déclaration des variables de requête et des variables de portée.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Alignez les clauses de requête sous la clause from, comme illustré dans les exemples précédents.
- Utilisez les clauses where avant les autres clauses de requête pour garantir que les clauses de requête ultérieures opèrent sur l'ensemble de données réduit et filtré.

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Utilisez plusieurs clauses `from` au lieu d'une clause join pour accéder aux collections internes. Par exemple, une collection d'objets `Student` peut contenir chacune une collection de scores de test. Lorsque la requête suivante est exécutée, elle retourne chaque score supérieur à 90, avec le nom de l'étudiant correspondant.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Sécurité

Suivez les instructions indiquées dans Instructions de codage sécurisé.

Voir aussi

- Conventions de codage Visual Basic
- Instructions de codage sécurisé

Main() et arguments de ligne de commande (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

La méthode `Main` est le point d'entrée d'une application C#. (Les bibliothèques et les services ne requièrent pas de `Main` méthode comme point d'entrée.) Lorsque l'application est démarrée, la `Main` méthode est la première méthode appelée.

Il ne peut y avoir qu'un seul point d'entrée dans un programme C#. Si vous avez plusieurs classes qui ont une `Main` méthode, vous devez compiler votre programme avec l' `-main` option de compilateur pour spécifier la `Main` méthode à utiliser comme point d'entrée. Pour plus d'informations, consultez -main (options du compilateur C#).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

Vue d'ensemble

- La méthode `Main` est le point d'entrée d'un programme exécutable ; c'est l'endroit où le contrôle du programme commence et se termine.
- `Main` est déclaré à l'intérieur d'une classe ou d'un struct. `Main` doit être statique et ne doit pas être public. (Dans l'exemple précédent, il reçoit l'accès par défaut de Private.) La classe ou le struct englobant ne doit pas obligatoirement être statique.
- `Main` peut avoir un type de retour `void`, `int` ou, à partir de C# 7.1, `Task` ou `Task<int>`.
- Si et seulement si `Main` retourne `Task` ou `Task<int>` , la déclaration de `Main` peut inclure le `async` modificateur. Notez que cela exclut spécifiquement une méthode `async void Main`.
- La méthode `Main` peut être déclarée avec ou sans paramètre `string[]`, qui contient des arguments de ligne de commande. Lorsque vous utilisez Visual Studio pour créer des applications Windows, vous pouvez ajouter le paramètre manuellement ou utiliser la GetCommandLineArgs() méthode pour obtenir les arguments de ligne de commande. Les paramètres sont lus comme des arguments de ligne de commande avec index de base zéro. Contrairement à C et C++, le nom du programme n'est pas traité comme le premier argument de ligne de commande dans le `args` tableau, mais il s'agit du premier élément de la GetCommandLineArgs() méthode.

La liste suivante répertorie les signatures valides `Main` :

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

Les exemples précédents utilisent tous le modificateur d'accesseur public. Cela est courant, mais pas obligatoire.

L'ajout des types de retour `async` et `Task`, `Task<int>` simplifie le code de programme lorsque les applications de console doivent démarrer et `await` des opérations asynchrones dans `Main`.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Génération à partir de la ligne de commande avec csc.exe
- Guide de programmation C#
- Méthodes
- À l'intérieur d'un programme C#

Arguments de ligne de commande (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

Vous pouvez envoyer des arguments à la méthode `Main` en définissant la méthode de l'une des manières suivantes :

```
static int Main(string[] args)
```

```
static void Main(string[] args)
```

> **NOTE**
> Pour activer des arguments de ligne de commande dans la `Main` méthode d'une application Windows Forms, vous devez modifier manuellement la signature de `Main` dans *Program.cs*. Le code généré par le Concepteur Windows Forms crée un `Main` sans paramètre d'entrée. Vous pouvez également utiliser Environment.CommandLine ou Environment.GetCommandLineArgs pour accéder aux arguments de ligne de commande à partir de n'importe quel emplacement d'une application console ou Windows.

Le paramètre de la méthode `Main` est un tableau String qui représente les arguments de ligne de commande. En général, vous déterminez s'il existe des arguments en testant la propriété `Length`, par exemple :

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

> **TIP**
> Le `args` tableau ne peut pas être null. Par conséquent, il est possible d'accéder à la `Length` propriété sans vérification de valeur null.

Vous pouvez également convertir les arguments de chaîne en types numériques à l'aide de la classe Convert ou de la méthode `Parse`. Par exemple, l'instruction suivante convertit `string` en nombre `long` en utilisant la méthode Parse :

```
long num = Int64.Parse(args[0]);
```

Il est également possible d'utiliser le type C# `long`, qui sert d'alias à `Int64` :

```
long num = long.Parse(args[0]);
```

Vous pouvez également utiliser la méthode `ToInt64` de la classe `Convert` pour obtenir le même résultat :

```
long num = Convert.ToInt64(s);
```

Pour plus d'informations, consultez Parse et Convert.

Exemple

L'exemple suivant montre comment utiliser des arguments de ligne de commande dans une application console. L'application prend un argument au moment de l'exécution, le convertit en entier, puis calcule la factorielle du nombre. Si aucun argument n'est fourni, l'application affiche un message pour expliquer comment le programme doit être utilisé.

Pour compiler et exécuter l'application à partir d'une invite de commandes, procédez comme suit :

1. Collez le code suivant dans un éditeur de texte, puis enregistrez le fichier en tant que fichier texte sous le nom *Factorial.cs*.

```
// Add a using directive for System if the directive isn't already present.

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. Dans l'écran **Démarrer** ou le menu **Démarrer**, ouvrez une fenêtre **Invite de commandes développeur** Visual Studio, puis accédez au dossier qui contient le fichier que vous venez de créer.

3. Entrez la commande suivante pour compiler l'application.

```
csc Factorial.cs
```

Si votre application n'a pas d'erreurs de compilation, un fichier exécutable nommé *Factorial.exe* est créé.

4. Entrez la commande suivante pour calculer la factorielle de 3 :

```
Factorial 3
```

5. Cette commande génère la sortie suivante : `The factorial of 3 is 6.`

> **NOTE**
> Quand vous exécutez une application dans Visual Studio, vous pouvez spécifier des arguments de ligne de commande dans la page Déboguer du Concepteur de projet.

Voir aussi

- System.Environment
- Guide de programmation C#
- Main () et arguments de ligne de commande
- Comment afficher les arguments de ligne de commande
- Valeurs de retour Main()
- Classes

Comment afficher des arguments de ligne de commande (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Les arguments fournis à un fichier exécutable sur la ligne de commande sont accessibles via un paramètre facultatif à `Main` . Les arguments sont fournis sous la forme d'un tableau de chaînes. Chaque élément du tableau contient un argument. Les espaces blancs entre arguments sont supprimés. Par exemple, considérez ces appels de ligne de commande d'un fichier exécutable fictif :

ENTRÉE SUR LA LIGNE DE COMMANDE	TABLEAU DE CHAÎNES PASSÉ À MAIN
executable.exe a b c	"a" "b" "c"
executable.exe un deux	"un" "deux"
executable.exe "un deux" trois	"un deux" "trois"

> **NOTE**
> Quand vous exécutez une application dans Visual Studio, vous pouvez spécifier des arguments de ligne de commande dans la page Déboguer du Concepteur de projets.

Exemple

Cet exemple affiche les arguments de ligne de commande passés à une application de ligne de commande. La sortie présentée concerne la première entrée du tableau ci-dessus.

```
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements.
        Console.WriteLine($"parameter count = {args.Length}");

        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine($"Arg[{i}] = [{args[i]}]");
        }
    }
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

Voir aussi

- Guide de programmation C#
- Génération à partir de la ligne de commande avec csc.exe
- Main () et arguments de ligne de commande
- Valeurs de retour Main()

Valeurs de retour de Main() (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

La méthode `Main` peut retourner `void` :

```
static void Main()
{
    //...
}
```

Elle peut également retourner un `int` :

```
static int Main()
{
    //...
    return 0;
}
```

Si la valeur de retour de `Main` n'est pas utilisée, retourner `void` permet d'avoir un code un peu plus simple. Cependant, retourner un entier permet au programme de communiquer des informations d'état à d'autres programmes ou scripts qui appellent le fichier exécutable. La valeur de retour de `Main` est traitée comme le code de sortie du processus. Si `void` est retourné à partir de `Main` , le code de sortie est implicitement `0` . L'exemple suivant montre comment accéder à la valeur de retour de `Main`.

Exemple

Cet exemple utilise les outils en ligne de commande .net Core . Si vous n'êtes pas familiarisé avec les outils en ligne de commande .NET Core, vous pouvez en savoir plus à ce sujet dans cet article de prise en main.

Modifiez la méthode `Main` dans *program.cs* comme suit :

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

Quand un programme est exécuté dans Windows, toute valeur retournée par la fonction `Main` est stockée dans une variable d'environnement. Cette variable d'environnement peut être récupérée à l'aide `ERRORLEVEL` de à partir d'un fichier de commandes ou `$LastExitCode` de PowerShell.

Vous pouvez générer l'application à l'aide de la commande dotnet CLI `dotnet build`.

Ensuite, créez un script PowerShell pour exécuter l'application et afficher le résultat. Collez le code suivant dans un fichier texte et enregistrez-le sous `test.ps1` dans le dossier qui contient le projet. Exécutez le script PowerShell en tapant `test.ps1` à l'invite de PowerShell.

Comme le code retourne zéro, le fichier de commandes indique la réussite. Toutefois, si vous modifiez MainReturnValTest.cs pour retourner une valeur différente de zéro, puis recompilez le programme, l'exécution suivante du script PowerShell signalera un échec.

```
dotnet run
```

```
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

Exemple de sortie

```
Execution succeeded
Return value = 0
```

Valeurs de retour d'Async Main

Les valeurs de retour d'Async Main déplacent le code réutilisable nécessaire pour appeler les méthodes asynchrones dans `Main` vers le code généré par le compilateur. Avant, vous auriez dû écrire cette construction pour appeler du code asynchrone et vérifier que votre programme s'est exécuté jusqu'à la fin de l'opération asynchrone :

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Maintenant, vous pouvez la remplacer par :

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

L'avantage de la nouvelle syntaxe est que le compilateur génère toujours un code correct.

Code généré par le compilateur

Quand le point d'entrée de l'application retourne `Task` ou `Task<int>`, le compilateur génère un nouveau point d'entrée qui appelle la méthode de point d'entrée déclarée dans le code d'application. En supposant que ce point d'entrée est appelé `$GeneratedMain`, le compilateur génère le code suivant pour ces points d'entrée :

- `static Task Main()` permet au compilateur d'émettre l'équivalent de

`private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`

- `static Task Main(string[])` permet au compilateur d'émettre l'équivalent de `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` permet au compilateur d'émettre l'équivalent de `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` permet au compilateur d'émettre l'équivalent de `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

> **NOTE**
>
> Si les exemples ont utilisé le modificateur `async` sur la méthode `Main`, le compilateur génère le même code.

Voir aussi

- Guide de programmation C#
- Référence C#
- Main () et arguments de ligne de commande
- Comment afficher les arguments de ligne de commande

Concepts de programmation (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette section explique les concepts de programmation du langage C#.

Dans cette section

TITRE	DESCRIPTION
Assemblys dans .NET	Explique comment créer et utiliser des assemblys.
Programmation asynchrone avec Async et Await (C#)	Explique comment écrire des solutions asynchrones à l'aide des mots clés async et await en C#. Comprend une procédure pas à pas.
Attributs (C#)	Explique comment fournir des informations supplémentaires sur des éléments de programmation tels que les types, les champs, les méthodes et les propriétés à l'aide d'attributs.
Collections (C#)	Décrit quelques-uns des types de collections fournis par .NET. Montre comment utiliser des collections simples et des collections de paires clé/valeur.
Covariance et contravariance (C#)	Montre comment activer la conversion implicite de paramètres de type générique dans les interfaces et les délégués.
Arborescences d'expressions (C#)	Explique comment utiliser des arborescences d'expressions pour activer la modification dynamique de code exécutable.
Itérateurs (C#)	Décrit les itérateurs, utilisés pour parcourir les collections et retourner les éléments un par un.
LINQ (Language-Integrated Query) (C#)	Décrit les puissantes capacités de requêtage de la syntaxe du langage C# et le modèle d'interrogation de bases de données relationnelles, de documents XML, de jeux de données et de collections en mémoire.
Réflexion (C#)	Explique comment utiliser la réflexion pour créer dynamiquement une instance d'un type, pour lier le type à un objet existant ou pour obtenir le type d'un objet existant, et comment invoquer ses méthodes ou accéder à ses champs et propriétés.
Sérialisation (C#)	Décrit les concepts clés de la sérialisation binaire, XML et SOAP.

Sections connexes

Conseils sur les performances	Décrit plusieurs règles de base qui peuvent aider à augmenter les performances des applications.

Programmation asynchrone avec Async et Await

02/11/2020 • 23 minutes to read • Edit Online

Le modèle de programmation asynchrone de tâche (TAP) fournit une abstraction sur le code asynchrone. Pour cela, vous écrivez votre code comme d'habitude, sous la forme d'une suite d'instructions. Vous pouvez lire ce code comme si chaque instruction se terminait avant que la suivante ne commence. Le compilateur effectue un certain nombre de transformations, car certaines de ces instructions peuvent commencer le travail et retourner un Task qui représente le travail en cours.

C'est en fait l'objectif de cette syntaxe : permettre au code d'être lu comme une suite d'instructions, mais d'être exécuté dans un ordre beaucoup plus complexe, en fonction de l'allocation des ressources externes et du moment auquel se terminent les tâches. Cela revient à donner des instructions pour des processus qui comprennent des tâches asynchrones. Tout au long de cet article, vous allez utiliser un exemple d'instructions pour créer un petit-déjeuner afin de voir comment les mots clés et facilitent la création de `async` `await` code, qui comprend une série d'instructions asynchrones. Pour expliquer comment préparer un petit-déjeuner, vous pourriez écrire des instructions telles que les suivantes :

1. Verser le café dans une tasse.
2. Faire chauffer la poêle, puis faire cuire deux œufs au plat.
3. Faire frire trois tranches de bacon.
4. Faire toaster deux tranches de pain.
5. Étaler le beurre et la confiture sur les toasts.
6. Verser du jus d'orange dans un verre.

Si vous avez l'habitude de cuisiner, vous savez que ces instructions doivent être exécutées de manière **asynchrone**. En effet, vous commencez par faire chauffer la poêle pour les œufs, mais vous faites d'abord frire le bacon. Ensuite, vous mettez les tranches de pain dans le grille-pain, puis vous cassez les œufs dans la poêle. À chaque étape du processus, vous démarrez une tâche, puis déplacez votre attention vers les tâches qui sont prêtes à être réalisées.

La préparation du petit-déjeuner est un bon exemple de travail asynchrone mais non parallèle. Une personne (ou un thread) peut gérer toutes ces tâches. Pour poursuivre l'analogie avec le petit-déjeuner, une personne peut préparer le petit-déjeuner de façon asynchrone en démarrant une tâche avant que la précédente ne soit terminée. En outre, la cuisson se poursuit, que vous la surveilliez ou non. Dès que vous commencez à faire chauffer la poêle pour les œufs, vous pouvez mettre le bacon à frire. Quand le bacon est en train de frire, vous pouvez mettre les tranches de pain dans le grille-pain.

Pour un algorithme parallèle, vous auriez besoin de plusieurs cuisiniers (ou threads). L'un d'eux s'occuperait des œufs, un autre du bacon, etc. Chacun d'eux serait concentré sur une seule tâche. Chaque cuisinier (ou thread) serait bloqué de façon synchrone, car il devrait attendre que le bacon soit prêt à être retourné ou que le pain soit grillé.

Écrivons maintenant ces instructions sous la forme d'instructions C# :

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");
```

```
            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

Le petit déjeuner préparé de façon synchrone a duré environ 30 minutes, car le total est la somme de chaque tâche individuelle.

> **NOTE**
> Les `Coffee` classes,,, `Egg` `Bacon` `Toast` et `Juice` sont vides. Il s'agit simplement de classes de marqueurs à des fins de démonstration, ne contiennent aucune propriété et n'a aucune autre fonction.

Les ordinateurs n'interprètent pas ces instructions de la même façon que les humains. L'ordinateur se bloque à chaque instruction jusqu'à ce que le travail soit terminé, avant de passer à l'instruction suivante. Avec une telle manière de procéder, notre petit-déjeuner risque de ne pas être très satisfaisant. En effet, chacune des tâches ne pourrait être démarrée qu'une fois la précédente terminée. De cette façon, la préparation du petit-déjeuner prendrait beaucoup plus de temps, et certains aliments refroidiraient avant d'être servis.

Si vous souhaitez que l'ordinateur exécute les instructions ci-dessus de façon asynchrone, vous devez écrire du code asynchrone.

De nos jours, ces remarques sont très importantes pour l'écriture de programmes. Lorsque vous écrivez des programmes clients, l'interface utilisateur doit être réactive aux entrées utilisateur. Votre application ne doit pas donner l'impression que votre smartphone se bloque pour télécharger des données à partir du web. Lorsque vous écrivez des programmes de serveur, vous ne devez pas bloquer les threads. En effet, ces threads peuvent servir à d'autres requêtes. Lorsque d'autres alternatives sont possibles, il est dommage d'utiliser du code synchrone, car celui-ci ne vous permet pas de réduire les frais des scale-out. Les threads bloqués ont un

prix.

Une application moderne efficace a besoin de code asynchrone. Avant la prise en charge du langage, l'écriture de code asynchrone nécessitait des rappels, des événements de complétion ou d'autres méthodes qui masquaient l'objectif premier du code. L'avantage du code synchrone est qu'il s'agit d'actions pas à pas qui facilitent l'analyse et la compréhension. Les modèles asynchrones traditionnels vous obligeaient à vous concentrer sur la nature asynchrone du code, et non sur ses actions fondamentales.

Éviter les blocages avec Await

Le code précédent montre une pratique déconseillée : écrire du code synchrone pour effectuer des opérations asynchrones. Comme nous l'avons vu, ce code bloque le thread, qui ne peut pas exécuter d'autres tâches. Il ne sera pas interrompu tant que l'une des tâches est en cours d'exécution. Ce serait comme rester les yeux fixés sur le grille-pain après avoir mis le pain dedans. Vous n'écouteriez personne tant que le pain ne serait pas ressorti du grille-pain.

Commençons par mettre à jour ce code pour que le thread ne se bloque pas pendant l'exécution des tâches. Le mot clé `await` permet de démarrer une tâche sans bloquer le thread, puis de poursuivre l'exécution une fois cette tâche terminée. La version asynchrone du code de préparation du petit-déjeuner ressemblerait à l'extrait de code suivant :

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

IMPORTANT

Le temps total écoulé est à peu près identique à la version synchrone initiale. Le code n'a pas encore pu tirer parti de certaines des fonctionnalités clés de la programmation asynchrone.

TIP

Les corps de méthode des `FryEggsAsync`, `FryBaconAsync` et `ToastBreadAsync` ont tous été mis à jour pour retourner `Task<Egg>`, `Task<Bacon>` et `Task<Toast>` respectivement. Les méthodes sont renommées à partir de leur version d'origine pour inclure le suffixe « Async ». Leurs implémentations sont indiquées dans le cadre de la version finale plus loin dans cet article.

Ce code ne se bloque pas pendant la cuisson des œufs ou du bacon. Cependant, il ne démarre pas d'autres tâches. Cela reviendrait, une nouvelle fois, à rester les yeux fixés sur le grille-pain en attendant que le pain

ressorte. Mais au moins, vous seriez en mesure de répondre à quelqu'un qui aurait besoin de votre attention. Dans un restaurant qui doit gérer plusieurs commandes en même temps, le chef peut commencer un petit-déjeuner pendant qu'un autre est en cours de préparation.

Ici, le thread qui prépare le petit-déjeuner n'est pas bloqué en attendant qu'une tâche démarrée se termine. Pour certaines applications, cette modification est la seule nécessaire. Ce simple changement permet à une application GUI de continuer à répondre à l'utilisateur. Toutefois, pour ce scénario, d'autres modifications sont nécessaires. Il n'est pas souhaitable que chaque tâche de composant soit exécutée de manière séquentielle. Il est préférable de démarrer chacune des tâches de composant avant d'attendre l'achèvement de la tâche précédente.

Démarrer plusieurs tâches simultanément

Dans de nombreux scénarios, il est nécessaire de démarrer plusieurs tâches immédiatement. Ensuite, à la fin de chaque tâche, vous pouvez passer aux autres tâches qui sont prêtes. C'est de cette façon que nous pouvons accélérer la préparation de notre petit-déjeuner. Cela permet également de tout terminer en même temps, ou presque. Cette façon de procéder vous garantit que vous mangerez votre petit-déjeuner bien chaud.

System.Threading.Tasks.Task et les types associés sont des classes que vous pouvez utiliser pour organiser les tâches en cours. Cela vous permet d'écrire du code qui se rapproche davantage de la façon dont vous devez préparer un petit-déjeuner, puisque vous commencez à faire cuire les œufs, à faire frire le bacon et à faire griller le pain simultanément. Puisque toutes ces tâches nécessitent une action, vous devez porter votre attention sur une tâche, démarrer l'action suivante, puis attendre qu'une autre tâche nécessite votre attention.

Vous démarrez une tâche puis vous gardez l'objet Task qui représente le travail. Vous devez attendre (`await`) chaque tâche avant d'utiliser ses résultats.

À présent, apportons ces modifications au code de notre petit-déjeuner. La première étape consiste à stocker les tâches des opérations dès leur démarrage, plutôt que d'attendre la fin de leur exécution :

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");

Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Ensuite, vous pouvez déplacer les instructions `await` pour le bacon et les œufs à la fin de la méthode, avant de servir le petit-déjeuner :

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

Le petit déjeuner préparé de façon asynchrone a duré environ 20 minutes, car certaines tâches pouvaient s'exécuter simultanément.

Le code précédent est plus efficace. Vous y démarrez toutes les tâches asynchrones en même temps. Vous n'attendez la fin d'une tâche que si vous avez besoin des résultats. Le code précédent est semblable au code des applications web qui envoient des requêtes à différents microservices, puis qui combinent les résultats au sein d'une même page. Vous allez exécuter toutes les requêtes en même temps, puis vous allez attendre (`await`) toutes ces tâches pour composer la page web.

Composition de tâches

Tout est prêt en même temps pour votre petit-déjeuner, sauf les toasts. La préparation des toasts correspond à la composition d'une opération asynchrone (faire griller le pain) et d'opérations synchrones (ajouter du beurre et de la confiture). Le code ci-dessous illustre un concept important :

> **IMPORTANT**
> La composition d'une opération asynchrone et d'une tâche synchrone constitue une opération asynchrone. Autrement dit, si une partie d'une opération est asynchrone, cela rend asynchrone l'intégralité de l'opération.

Le code précédent montre que vous pouvez utiliser les objets Task ou Task<TResult> pour attendre les tâches en cours d'exécution. Vous attendez (`await`) la fin de chaque tâche avant d'utiliser ses résultats. L'étape suivante consiste à créer des méthodes qui représentent la combinaison d'autres tâches. Avant de servir le petit-déjeuner, vous devez attendre la fin de la tâche qui correspond à l'action de faire griller le pain, avant d'ajouter le beurre et la confiture. Vous pouvez représenter cette tâche à l'aide du code suivant :

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

La signature de la méthode précédente comprend le modificateur `async`. Cela signale au compilateur que cette méthode contient une instruction `await`. Elle contient des opérations asynchrones. Cette méthode représente la tâche qui consiste à faire griller le pain, puis à ajouter le beurre et la confiture. Cette méthode retourne un Task<TResult> qui représente la composition de ces trois opérations. Le bloc de code principal devient alors :

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

La modification précédente montre une technique importante pour l'utilisation du code asynchrone. Pour composer des tâches, vous devez séparer les opérations dans une nouvelle méthode qui retourne une tâche. Vous pouvez choisir dans quels cas attendre les tâches. Vous pouvez démarrer d'autres tâches simultanément.

Attendre efficacement la fin des tâches

La série d'instructions `await` à la fin du code précédent peut être améliorée à l'aide des méthodes de la classe `Task`. L'une de ces API est WhenAll, qui retourne un Task se terminant lorsque toutes les tâches de sa liste

d'arguments sont terminées, comme indiqué dans le code suivant :

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Une autre option consiste à utiliser WhenAny, qui retourne un `Task<Task>` se terminant lorsque l'un de ses arguments se termine. Vous pouvez attendre la tâche retournée, tout en sachant qu'elle est déjà terminée. Le code suivant montre comment utiliser WhenAny pour attendre la fin de la première tâche, puis traiter ses résultats. Après avoir traité les résultats de la tâche terminée, vous pouvez la supprimer de la liste des tâches passée à `WhenAny`.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

Après toutes ces modifications, la version finale du code ressemble à ceci :

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
```

```
            }
            else if (finishedTask == toastTask)
            {
                Console.WriteLine("toast is ready");
            }
            breakfastTasks.Remove(finishedTask);
        }

        Juice oj = PourOJ();
        Console.WriteLine("oj is ready");
        Console.WriteLine("Breakfast is ready!");
    }

    static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
    {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);

        return toast;
    }

    private static Juice PourOJ()
    {
        Console.WriteLine("Pouring orange juice");
        return new Juice();
    }

    private static void ApplyJam(Toast toast) =>
        Console.WriteLine("Putting jam on the toast");

    private static void ApplyButter(Toast toast) =>
        Console.WriteLine("Putting butter on the toast");

    private static async Task<Toast> ToastBreadAsync(int slices)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        Console.WriteLine("Start toasting...");
        await Task.Delay(3000);
        Console.WriteLine("Remove toast from toaster");

        return new Toast();
    }

    private static async Task<Bacon> FryBaconAsync(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the pan");
        Console.WriteLine("cooking first side of bacon...");
        await Task.Delay(3000);
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        await Task.Delay(3000);
        Console.WriteLine("Put bacon on plate");

        return new Bacon();
    }

    private static async Task<Egg> FryEggsAsync(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        await Task.Delay(3000);
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
```

```
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

La version finale du petit-déjeuner préparé de façon asynchrone a duré environ 15 minutes. cela est dû au fait que certaines tâches pouvaient s'exécuter simultanément et que le code était en mesure de surveiller plusieurs tâches à la fois et d'agir uniquement lorsque cela était nécessaire.

Le code final est asynchrone. Il reflète plus précisément la façon dont nous préparons habituellement le petit-déjeuner. Comparez le code précédent au premier exemple de code de cet article. Les actions de base sont toujours claires lorsque nous lisons le code. Ce code est tout aussi lisible que les instructions de préparation du petit-déjeuner au début de cet article. Les fonctionnalités de langage pour `async` et `await` traduisent ce que chaque personne fait lorsqu'elle suit des instructions : démarrer les tâches dès que possible et ne pas attendre qu'une tâche soit terminée avant de passer à la suivante.

Étapes suivantes

En savoir plus sur le modèle de programmation asynchrone de tâche

Modèle de programmation asynchrone des tâches

02/11/2020 • 27 minutes to read • Edit Online

Vous pouvez éviter des goulots d'étranglement de performance et améliorer la réactivité globale de votre application à l'aide de la programmation asynchrone. Toutefois, les techniques traditionnelles pour écrire des applications asynchrones peuvent être complexes et rendre ces applications difficiles à écrire, déboguer et mettre à jour.

C# 5 a introduit une approche simplifiée, la programmation asynchrone, qui tire parti de la prise en charge asynchrone de .NET Framework 4.5 et des versions ultérieures, de .NET Core, ainsi que de Windows Runtime. Le compilateur effectue le travail difficile dont se chargeait le développeur jusqu'à maintenant. En outre, votre application conserve une structure logique qui ressemble à du code synchrone. Par conséquent, vous obtenez tous les avantages de la programmation asynchrone avec peu d'effort.

Cette rubrique fournit une vue d'ensemble sur quand et comment utiliser la programmation asynchrone, et inclut des liens vers des rubriques du support, qui contiennent des informations et des exemples.

Async améliore la réactivité

Le comportement asynchrone est essentiel pour les activités qui sont potentiellement bloquantes, par exemple l'accès au web. L'accès à une ressource Web est parfois lent ou différé. Si cette activité est bloquée dans un processus synchrone, toute l'application doit attendre. Dans un processus asynchrone, l'application peut poursuivre une autre tâche qui ne dépend pas de la ressource Web jusqu'à ce que la tâche potentiellement bloquante soit terminée.

Le tableau suivant indique les zones classiques où la programmation asynchrone améliore la réactivité. Les API répertoriées de .NET et de Windows Runtime contiennent des méthodes qui prennent en charge la programmation asynchrone.

DOMAINE D'APPLICATION	TYPES .NET AVEC LES MÉTHODES ASYNC	TYPES WINDOWS RUNTIME AVEC LES MÉTHODES ASYNC
Accès Web	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Utilisation de fichiers	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
Utilisation des images		MediaCapture BitmapEncoder BitmapDecoder
Programmation WCF	Opérations synchrones et asynchrones	

Le comportement asynchrone est particulièrement utile pour les applications qui accèdent au thread d'interface utilisateur, car toute activité liée à l'interface utilisateur partage généralement un thread. Si un processus est bloqué dans une application synchrone, tous les processus sont bloqués. Votre application ne répond plus et vous pouvez conclure qu'elle a rencontré une défaillance, alors qu'elle attend simplement.

Lorsque vous utilisez des méthodes asynchrones, l'application continue à répondre à l'interface utilisateur. Vous pouvez redimensionner ou réduire une fenêtre, par exemple, ou vous pouvez fermer l'application si vous ne souhaitez pas attendre qu'elle se termine.

L'approche basée sur async ajoute l'équivalent d'une transmission automatique à la liste d'options dont vous disposez pour concevoir des opérations asynchrones. En d'autres termes, vous obtenez tous les avantages de la programmation asynchrone classique mais avec beaucoup moins d'efforts du point de vue du développeur.

Les méthodes Async sont faciles à écrire

Les mots clés async et await en C# sont au cœur de la programmation async. En utilisant ces deux mots clés, vous pouvez utiliser des ressources dans .NET Framework, .NET Core ou le Windows Runtime pour créer une méthode asynchrone presque aussi facilement que vous créez une méthode synchrone. Les méthodes asynchrones définies avec mot clé `async` sont appelées *méthodes asynchrones*.

L'exemple suivant illustre une méthode async. Presque tous les éléments du code doivent vous sembler familiers.

Vous trouverez un exemple complet de Windows Presentation Foundation (WPF) disponible en téléchargement à partir de la programmation asynchrone avec Async et await en C#.

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Vous pouvez tirer plusieurs enseignements à partir de l'exemple précédent. Démarrez par la signature de la méthode. Elle inclut le modificateur `async`. Le type de retour est `Task<int>` (consultez la section « Types de retour » pour découvrir d'autres options). Le nom de la méthode se termine dans `Async`. Dans le corps de la méthode, `GetStringAsync` retourne un `Task<string>`. Cela signifie que quand vous attendez (`await`) la tâche, vous obtenez un `string` (`contents`). Avant d'attendre la tâche, vous pouvez effectuer tout travail qui ne repose pas sur le `string` de `GetStringAsync`.

Faites particulièrement attention à l'opérateur `await`. Il s'interrompt `GetUrlContentLengthAsync` :

- `GetUrlContentLengthAsync` ne peut pas continuer tant que `getStringTask` n'est pas terminé.
- Pendant ce temps, le contrôle retourne à l'appelant de `GetUrlContentLengthAsync`.
- Le contrôle reprend ici quand `getStringTask` est terminé.
- L'opérateur `await` récupère alors le résultat `string` auprès de `getStringTask`.

L'instruction return spécifie un résultat entier. Toutes les méthodes qui attendent `GetUrlContentLengthAsync` récupèrent la valeur de longueur.

Si `GetUrlContentLengthAsync` n'a aucun travail qu'il peut effectuer entre l'appel de `GetStringAsync` et l'attente de son achèvement, vous pouvez simplifier votre code en appelant et attendant dans l'instruction unique suivante.

```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

Les caractéristiques suivantes résument ce qui fait de l'exemple précédent une méthode Async :

- La signature de la méthode inclut un modificateur `async`.
- Le nom d'une méthode async, par convention, se termine par un suffixe « Async ».
- Le type de retour est l'un des types suivants :
  - Task<TResult> si votre méthode a une instruction return dans laquelle l'opérande est de type `TResult`.
  - Task si votre méthode n'a aucune instruction de retour, ou si elle a une instruction de retour sans opérande.
  - `void` si vous écrivez un gestionnaire d'événements async.
  - Tous les autres types ayant une méthode `GetAwaiter` (à compter de C# 7.0).

  Pour plus d'informations, consultez la section types de retour et paramètres .
- La méthode inclut généralement au moins une expression `await`, qui marque le point au-delà duquel la méthode ne peut pas poursuivre son exécution tant que l'opération asynchrone attendue n'est pas terminée. Dans le même temps, la méthode est interrompue, et le contrôle retourne à l'appelant de la méthode. La section suivante de cette rubrique illustre ce qui se produit au point d'interruption.

Dans les méthodes async, vous utilisez les mots clés et les types fournis pour indiquer ce que vous souhaitez faire, et le compilateur effectue le reste, notamment le suivi de ce qui doit se produire lorsque le contrôle retourne à un point d'attente dans une méthode interrompue. Il peut être difficile de gérer des processus de routine, tels que les boucles et la gestion des exceptions, dans le code asynchrone traditionnel. Dans une méthode async, vous écrivez ces éléments comme vous l'auriez fait dans une solution synchrone, et le problème est résolu.

Pour plus d'informations sur l'asynchronie dans les versions précédentes de .NET Framework, consultez tpl et programmation asynchrone .NET Framework traditionnelle.

Ce qui se passe dans une méthode Async

La chose la plus importante à comprendre en programmation asynchrone est le déplacement du flux de contrôle d'une méthode à l'autre. Le diagramme suivant présente le processus :

Les nombres dans le diagramme correspondent aux étapes suivantes, initiées lorsqu'une méthode d'appel appelle la méthode Async.

1. Une méthode d'appel appelle et attend la `GetUrlContentLengthAsync` méthode Async.

2. `GetUrlContentLengthAsync` crée une instance HttpClient et appelle la méthode asynchrone GetStringAsync pour télécharger le contenu d'un site Web comme une chaîne.

3. Quelque chose se produit dans `GetStringAsync` et suspend sa progression. Elle peut être obligée d'attendre la fin d'un téléchargement sur un site Web ou de toute autre activité bloquante. Pour éviter de bloquer les ressources, `GetStringAsync` cède le contrôle à son appelant, `GetUrlContentLengthAsync`.

   `GetStringAsync` retourne un Task<TResult>, où `TResult` est une chaîne, et `GetUrlContentLengthAsync` assigne la tâche à la variable `getStringTask`. La tâche représente le processus en cours de l'appel de `GetStringAsync`, avec l'engagement de produire une valeur de chaîne réelle lorsque le travail est terminé.

4. Étant donné que `getStringTask` n'a pas encore été attendu, `GetUrlContentLengthAsync` peut continuer avec un autre travail qui ne dépend pas du résultat final issu de `GetStringAsync`. Cette opération est représentée par un appel à la méthode synchrone `DoIndependentWork`.

5. `DoIndependentWork` est une méthode synchrone qui effectue son travail et retourne à son appelant.

6. `GetUrlContentLengthAsync` n'a plus de travail à exécuter sans un résultat de `getStringTask`. `GetUrlContentLengthAsync` veut ensuite calculer et retourner la longueur de la chaîne téléchargée, mais la méthode ne peut pas calculer cette valeur avant d'avoir la chaîne.

   Par conséquent, `GetUrlContentLengthAsync` utilise un opérateur await pour interrompre sa progression et pour céder le contrôle à la méthode ayant appelé `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` retourne un `Task<int>` à l'appelant. La tâche représente la promesse de produire un résultat entier qui est la longueur de la chaîne téléchargée.

> **NOTE**
> Si `GetStringAsync` (et donc `getStringTask`) se termine avant que `GetUrlContentLengthAsync` ne l'attende, le contrôle reste dans `GetUrlContentLengthAsync`. Le coût de la suspension et du retour à est `GetUrlContentLengthAsync` gaspillé si le processus asynchrone appelé `getStringTask` est déjà terminé et `GetUrlContentLengthAsync` n'a pas besoin d'attendre le résultat final.

À l'intérieur de la méthode d'appel, le modèle de traitement continue. L'appelant peut effectuer d'autres tâches qui ne dépendent pas du résultat de `GetUrlContentLengthAsync` avant d'attendre ce résultat, ou l'appelant peut attendre immédiatement. La méthode appelante attend `GetUrlContentLengthAsync`, et `GetUrlContentLengthAsync` attend `GetStringAsync`.

7. `GetStringAsync` se termine et génère un résultat de chaîne. Le résultat de chaîne n'est pas retourné par l'appel de `GetStringAsync` de la façon à laquelle vous pourriez vous attendre. (N'oubliez pas que la méthode a déjà retourné une tâche à l'étape 3.) Au lieu de cela, le résultat chaîne est stocké dans la tâche qui représente l'achèvement de la méthode, `getStringTask`. L'opérateur await récupère le résultat de `getStringTask`. L'instruction d'assignation assigne le résultat récupéré à `contents`.

8. Lorsque `GetUrlContentLengthAsync` a le résultat de chaîne, la méthode peut calculer la longueur de la chaîne. Puis, le travail d' `GetUrlContentLengthAsync` est également interrompu, et le gestionnaire d'événements en attente peut reprendre. Dans l'exemple complet à la fin de la rubrique, vous pouvez vérifier que le gestionnaire d'événements récupère et imprime la valeur du résultat de la longueur. Si vous débutez en programmation asynchrone, prenez une minute pour déterminer la différence entre le comportement synchrone et le comportement asynchrone. Une méthode synchrone retourne une fois son travail terminé (étape 5), mais une méthode asynchrone retourne une valeur de tâche lorsque son travail est interrompu (étapes 3 et 6). Lorsque la méthode async termine finalement son travail, la tâche est marquée comme terminée et le résultat, le cas échéant, est stocké dans la tâche.

Méthodes Async d'API

Vous pouvez vous demander où rechercher les méthodes telles que `GetStringAsync` qui prennent en charge la programmation async. .NET Framework 4,5 ou version ultérieure et .NET Core contiennent de nombreux membres qui fonctionnent avec `async` et `await`. Vous pouvez les reconnaître par le suffixe « Async » qui est ajouté au nom de membre et par leur type de retour Task ou Task<TResult>. Par exemple, la classe `System.IO.Stream` contient des méthodes telles que CopyToAsync, ReadAsync et WriteAsync en même temps que les méthodes synchrones CopyTo, Read et Write.

Windows Runtime contient également de nombreuses méthodes utilisables avec `async` et `await` dans les applications Windows. Pour plus d'informations, consultez Programmation thread et asynchrone pour le développement UWP, et Programmation asynchrone (applications Windows Store) et Démarrage rapide : appel d'API asynchrones en C# ou Visual Basic si vous utilisez des versions antérieures de Windows Runtime.

Thèmes

Les méthodes Async sont conçues pour être des opérations non bloquantes. Une `await` expression dans une méthode Async ne bloque pas le thread actuel pendant que la tâche attendue est en cours d'exécution. Au lieu de cela, l'expression inscrit le reste de la méthode comme continuation et retourne le contrôle à l'appelant de la méthode async.

Les mots clés `async` et `await` n'entraînent pas la création de threads supplémentaires. Les méthodes Async ne requièrent pas de multithreading, car une méthode async ne fonctionne pas sur son propre thread. La méthode s'exécute sur le contexte de synchronisation actuel et utilise du temps sur le thread uniquement lorsqu'elle est active. Vous pouvez utiliser Task.Run pour déplacer le travail lié au processeur vers un thread d'arrière-plan, mais

un thread d'arrière-plan ne permet pas à un processus qui attend simplement les résultats de devenir disponible.

L'approche basée sur async en matière de programmation asynchrone est préférable aux approches existantes, dans presque tous les cas. En particulier, cette approche est préférable à la classe BackgroundWorker pour les opérations utilisant de nombreuses E/S, car le code est plus simple et il est inutile de vous protéger contre des conditions de concurrence. En association avec la Task.Run méthode, la programmation asynchrone est meilleure que BackgroundWorker pour les opérations liées au processeur, car la programmation asynchrone sépare les détails de coordination de l'exécution de votre code du travail qui `Task.Run` transfère vers le pool de threads.

Async et await

Si vous spécifiez qu'une méthode est une méthode async en utilisant le modificateur async, vous activez les deux fonctionnalités suivantes.

- La méthode async marquée peut utiliser await pour indiquer des points d'interruption. L'opérateur `await` indique au compilateur que la méthode asynchrone ne peut pas continuer au-delà de ce point, tant que le processus asynchrone attendu n'est pas terminé. Entre-temps, le contrôle retourne à l'appelant de la méthode async.

  La suspension d'une méthode Async au niveau d'une `await` expression ne constitue pas une sortie de la méthode et les `finally` blocs ne s'exécutent pas.

- La méthode async marquée peut elle-même être attendue par les méthodes qui l'appellent.

Une méthode Async contient généralement une ou plusieurs occurrences d'un `await` opérateur, mais l'absence d' `await` expressions ne provoque pas d'erreur du compilateur. Si une méthode Async n'utilise pas un `await` opérateur pour marquer un point de suspension, la méthode s'exécute en tant que méthode synchrone, en dépit du `async` modificateur. Le compilateur émet un avertissement pour ces méthodes.

`async` et `await` sont des mots clés contextuels. Pour plus d'informations et pour obtenir des exemples, consultez les rubriques suivantes :

- async
- await

Types de retour et paramètres

Une méthode async retourne généralement un Task ou un Task<TResult>. Dans une méthode async, un opérateur `await` est appliqué à une tâche retournée à partir d'un appel à une autre méthode async.

Vous spécifiez Task<TResult> comme type de retour si la méthode contient une `return` instruction qui spécifie un opérande de type `TResult` .

Vous utilisez Task comme type de retour si la méthode n'a aucune instruction return ou une instruction return qui ne retourne pas d'opérande.

À compter de C# 7.0, vous pouvez également spécifier n'importe quel autre type de retour, sous réserve que ce type inclue une méthode `GetAwaiter`. ValueTask<TResult> est un exemple d'un tel type. Il est disponible dans le package NuGet System.Threading.Tasks.Extension.

L'exemple suivant montre comment déclarer et appeler une méthode qui retourne un Task<TResult> ou un Task :

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Chaque tâche retournée représente le travail en cours. Une tâche encapsule des informations sur l'état du processus asynchrone et, éventuellement, le résultat final du processus ou l'exception que le processus déclenche s'il ne réussit pas.

Une méthode async peut également avoir un type de retour `void`. Ce type de retour est essentiellement utilisé pour définir les gestionnaires d'événements, où un type de retour `void` est obligatoire. Les gestionnaires d'événements asynchrones servent souvent de point de départ aux programmes asynchrones.

Une méthode Async qui a un `void` type de retour ne peut pas être attendue, et l'appelant d'une méthode retournant void ne peut pas intercepter les exceptions levées par la méthode.

Une méthode async ne peut pas déclarer de paramètres in, ref ou out, mais elle peut appeler des méthodes qui ont ces paramètres. De même, une méthode async ne peut pas retourner une valeur par référence, bien qu'elle puisse appeler des méthodes avec des valeurs de retour de référence.

Pour plus d'informations et d'exemples, consultez types de retour Async (C#). Pour plus d'informations sur l'interception des exceptions dans les méthodes async, consultez la page try-catch.

Les API asynchrones dans la programmation Windows Runtime ont l'un des types de retour suivants, qui sont semblables aux tâches :

- IAsyncOperation<TResult>, qui correspond à Task<TResult>
- IAsyncAction, qui correspond à Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Convention d'affectation de noms

Par Convention, les méthodes qui retournent des types habituellement await (par exemple,,, `Task` `Task<T>` `ValueTask` `ValueTask<T>`) doivent avoir des noms qui se terminent par « Async ». Les méthodes qui démarrent une opération asynchrone, mais qui ne retournent pas un type awaitable ne doivent pas avoir un nom qui se termine par « Async ». Cependant, leur nom peut commencer par « Begin », « Start » ou un autre mot suggérant que cette méthode ne retourne pas ou ne lève pas le résultat de l'opération.

Vous pouvez ignorer la convention où un événement, une classe de base, ou un contrat d'interface suggère un nom différent. Par exemple, vous ne devez pas renommer les gestionnaires d'événements communs, tels que `OnButtonClick` .

Rubriques connexes et exemples (Visual Studio)

TITRE	DESCRIPTION	EXEMPLE
Comment effectuer plusieurs requêtes Web en parallèle en utilisant Async et await (C#)	Explique comment démarrer plusieurs tâches en même temps.	Exemple Async : effectuer plusieurs requêtes web en parallèle
Types de retour Async (C#)	Illustre les types que les méthodes Async peuvent retourner et explique quand chaque type est approprié.	
Annuler les tâches avec un jeton d'annulation en tant que mécanisme de signalisation.	Indique comment ajouter les fonctionnalités suivantes à votre solution async : - Annuler une liste de tâches (C#) - Annuler des tâches après une période de temps (C#) - Traiter la tâche asynchrone à mesure qu'elle se termine (C#)	
Utilisation d'Async pour l'accès aux fichiers (C#)	Répertorie et explique les avantages de l'utilisation d'async et d'await pour accéder aux fichiers.	
Modèle asynchrone basé sur les tâches (TAP)	Décrit un modèle asynchrone, le modèle est basé sur les Task Task<TResult> types et.	
Vidéos Async sur Channel 9	Fournit des liens vers diverses vidéos sur la programmation asynchrone.	

Voir aussi

- async
- await
- Programmation asynchrone
- Vue d'ensemble Async

Types de retour Async (C#)

02/11/2020 • 14 minutes to read • Edit Online

Les méthodes async peuvent avoir les types de retour suivants :

- Task, pour une méthode async qui effectue une opération mais ne retourne aucune valeur.
- Task<TResult>, pour une méthode async qui retourne une valeur.
- `void`, pour un gestionnaire d'événements.
- À compter de C# 7.0, tout type ayant une méthode `GetAwaiter` accessible. L'objet retourné par la méthode `GetAwaiter` doit implémenter l'interface System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- À compter de C# 8,0, IAsyncEnumerable<T> , pour une méthode Async qui retourne un *flux asynchrone*.

Pour plus d'informations sur les méthodes Async, consultez programmation asynchrone avec Async et await (C#).

Il existe également plusieurs autres types spécifiques aux charges de travail Windows :

- DispatcherOperation, pour les opérations asynchrones limitées à Windows.
- IAsyncAction, pour les actions asynchrones dans UWP qui ne retournent pas de valeur.
- IAsyncActionWithProgress<TProgress>, pour les actions asynchrones dans UWP qui signalent la progression mais ne retournent pas de valeur.
- IAsyncOperation<TResult>, pour les opérations asynchrones dans UWP qui retournent une valeur.
- IAsyncOperationWithProgress<TResult,TProgress>, pour les opérations asynchrones dans UWP qui signalent la progression et retournent une valeur.

Type de retour de la tâche

Les méthodes async qui ne contiennent pas d'instruction `return` ou qui contiennent une instruction `return` ne retournant pas d'opérande ont généralement un type de retour Task. De telles méthodes retournent `void` si elles s'exécutent de façon synchrone. Si vous utilisez un type de retour Task pour une méthode async, une méthode d'appel peut utiliser un opérateur `await` pour suspendre l'achèvement de l'appelant jusqu'à ce que la méthode async soit terminée.

Dans l'exemple suivant, la `WaitAndApologizeAsync` méthode ne contient pas d' `return` instruction, donc la méthode retourne un Task objet. Le retour `Task` d'un permet `WaitAndApologizeAsync` à d'être attendu. Le Task type n'inclut `Result` pas de propriété, car il n'a pas de valeur de retour.

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

`WaitAndApologizeAsync` est attendue avec une instruction await au lieu d'une expression await, comme pour l'instruction d'appel d'une méthode synchrone retournant un type void. Dans ce cas, l'application d'un opérateur await ne génère pas de valeur. Pour clarifier les termes instruction et expression, reportez-vous au tableau ci-dessous :

TYPE AWAIT	EXEMPLE	TYPE
.	`await SomeTaskMethodAsync()`	Task
Expression	`T result = await SomeTaskMethodAsync<T>();`	Task<TResult>

Vous pouvez séparer l'appel à `WaitAndApologizeAsync` de l'application d'un opérateur await, comme le montre le code suivant. Cependant, n'oubliez pas qu'un type `Task` n'a pas de propriété `Result` et qu'aucune valeur n'est générée quand un opérateur await est appliqué à un type `Task`.

Le code suivant sépare l'appel à la méthode `WaitAndApologizeAsync` de l'attente de la tâche que retourne la méthode.

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Type de retour de la tâche <TResult>

Le Task<TResult> type de retour est utilisé pour une méthode Async qui contient une instruction Return dans laquelle l'opérande est `TResult` .

Dans l'exemple suivant, la `GetLeisureHoursAsync` méthode contient une `return` instruction qui retourne un entier. La déclaration de méthode doit donc spécifier un type de retour `Task<int>`. La FromResult méthode Async est un

espace réservé pour une opération qui retourne un DayOfWeek .

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

Quand la méthode `GetLeisureHoursAsync` est appelée à partir d'une expression await dans la méthode `ShowTodaysInfo`, cette expression récupère la valeur entière (la valeur de `leisureHours`) qui est stockée dans la tâche retournée par la méthode `GetLeisureHours`. Pour plus d'informations sur les expressions await, consultez await.

Vous pouvez mieux comprendre comment `await` récupère le résultat d'un `Task<T>` en séparant l'appel à `GetLeisureHoursAsync` de l'application de `await` , comme le montre le code suivant. Un appel à la méthode `GetLeisureHoursAsync` qui n'est pas immédiatement attendue retourne un type `Task<int>`, comme vous pourriez l'attendre de la déclaration de la méthode. La tâche est assignée à la variable `getLeisureHoursTask` dans l'exemple. Comme `getLeisureHoursTask` est un Task<TResult>, il contient une propriété Result de type `TResult`. Dans ce cas, `TResult` représente un type entier. Quand `await` est appliqué à `getLeisureHoursTask`, l'expression await prend pour la valeur le contenu de la propriété Result de `getLeisureHoursTask`. La valeur est assignée à la variable `ret`.

IMPORTANT

Result est une propriété bloquante. Si vous essayez d'y accéder avant la fin de sa tâche, le thread actif est bloqué tant que la tâche n'est pas terminée et que la valeur n'est pas disponible. Dans la plupart des cas, vous devez accéder à la valeur avec `await` au lieu d'accéder directement à la propriété.

L'exemple précédent a récupéré la valeur de la Result propriété pour bloquer le thread principal afin que la `Main` méthode puisse imprimer `message` dans la console avant la fin de l'application.

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Type de retour void

Vous utilisez le type de retour `void` dans les gestionnaires d'événements asynchrones, lesquels exigent un type de retour `void`. Pour les méthodes autres que les gestionnaires d'événements qui ne retournent pas de valeur, vous devez retourner Task à la place, car une méthode async qui retourne `void` ne peut pas être attendue. Tout appelant d'une telle méthode doit continuer à se terminer sans attendre la fin de la méthode Async appelée. L'appelant doit être indépendant des valeurs ou des exceptions générées par la méthode Async.

L'appelant d'une méthode Async qui retourne void ne peut pas intercepter les exceptions levées à partir de la méthode, et ces exceptions non gérées sont susceptibles de provoquer l'échec de votre application. Si une méthode qui retourne un Task ou Task<TResult> lève une exception, l'exception est stockée dans la tâche retournée. L'exception est levée une nouvelle fois lorsque la tâche est attendue. Par conséquent, veillez à ce qu'une méthode async capable de générer une exception ait le type de retour Task ou Task<TResult> et que les appels à la méthode soient attendus.

Pour plus d'informations sur la façon d'intercepter des exceptions dans les méthodes Async, consultez la section exceptions dans les méthodes Async de l'article try-catch .

L'exemple suivant illustre le comportement d'un gestionnaire d'événements asynchrones. Dans l'exemple de code, un gestionnaire d'événements asynchrones doit permettre au thread principal de savoir quand il se termine. Le thread principal peut donc attendre la fin d'un gestionnaire d'événements asynchrones avant de quitter le programme.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
```

```
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Types de retour async généralisés et ValueTask<TResult>

À compter de C# 7.0, une méthode async peut retourner tout type ayant une méthode `GetAwaiter` accessible.

Étant donné que Task et Task<TResult> sont des types référence, l'allocation de mémoire dans des chemins critiques pour les performances, en particulier quand les allocations se produisent dans des boucles serrées, peuvent nuire aux performances. La prise en charge de types de retour généralisés signifie que vous pouvez retourner un type valeur léger au lieu d'un type référence pour éviter des allocations de mémoire supplémentaires.

Le .NET fournit la structure System.Threading.Tasks.ValueTask<TResult> comme implémentation légère d'une valeur retournant des tâches généralisées. Pour utiliser le type System.Threading.Tasks.ValueTask<TResult>, vous devez ajouter le package NuGet `System.Threading.Tasks.Extensions` à votre projet. L'exemple suivant utilise la structure ValueTask<TResult> pour récupérer la valeur de deux dés.

```
using System;
using System.Threading.Tasks;

class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
//    Shaking dice...
//    You rolled 8
```

Flux asynchrones avec IAsyncEnumerable<T>

À compter de C# 8,0, une méthode Async peut retourner un *flux asynchrone*, représenté par IAsyncEnumerable<T> . Un flux asynchrone fournit un moyen d'énumérer les éléments lus à partir d'un flux lorsque des éléments sont générés dans des segments avec des appels asynchrones répétés. L'exemple suivant montre une méthode Async qui génère un flux asynchrone :

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
          Here is the second line of text.
          And there is one more for good measure.
          Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ', StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

L'exemple précédent lit les lignes d'une chaîne de façon asynchrone. Une fois chaque ligne lue, le code énumère

chaque mot de la chaîne. Les appelants énumèrent chaque mot à l'aide de l' `await foreach` instruction. La méthode attend quand elle doit lire de façon asynchrone la ligne suivante à partir de la chaîne source.

Voir aussi

- FromResult
- Traiter les tâches asynchrones terminées
- Programmation asynchrone avec Async et await (C#)
- async
- await

Annuler une liste de tâches (C#)

02/11/2020 • 7 minutes to read • Edit Online

Vous pouvez annuler une application console Async si vous ne souhaitez pas attendre qu'elle se termine. En suivant l'exemple de cette rubrique, vous pouvez ajouter une annulation à une application qui télécharge le contenu d'une liste de sites Web. Vous pouvez annuler de nombreuses tâches en associant l' CancellationTokenSource instance à chaque tâche. Si vous sélectionnez la touche `entrée` , vous annulez toutes les tâches qui ne sont pas encore terminées.

Ce didacticiel contient les sections suivantes :

- Création d'une application console .NET
- Écriture d'une application Async qui prend en charge l'annulation
- Démonstration de l'annulation de la signalisation

Prérequis

Ce didacticiel requiert les éléments suivants :

- .NET 5,0 ou kit de développement logiciel (SDK) ultérieur
- Environnement de développement intégré (IDE)
  - Nous vous recommandons Visual Studio, Visual Studio Code ou Visual Studio pour Mac

Créer un exemple d'application

Créez une application de console .NET Core. Vous pouvez en créer un à l'aide de la `dotnet new console` commande ou de Visual Studio. Ouvrez le fichier *Program.cs* dans votre éditeur de code favori.

Remplacer les instructions using

Remplacez les instructions using existantes par les déclarations suivantes :

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Ajouter des champs

Dans la `Program` définition de classe, ajoutez les trois champs suivants :

```
static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

Le CancellationTokenSource est utilisé pour signaler une annulation demandée à un CancellationToken . Le `HttpClient` expose la possibilité d'envoyer des requêtes http et de recevoir des réponses http. Le `s_urlList` contient toutes les URL que l'application envisage de traiter.

Mettre à jour le point d'entrée de l'application

Le point d'entrée principal dans l'application console est la `Main` méthode. Remplacez la méthode existante par le code suivant :

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}
```

La méthode mise à jour `Main` est maintenant considérée comme une main asynchrone, ce qui permet d'obtenir un point d'entrée asynchrone dans l'exécutable. Il écrit quelques messages d'instructions dans la console, puis déclare une Task instance nommée `cancelTask` , qui lira les séquences de touches de la console. Si la touche entrée est enfoncée, un appel à CancellationTokenSource.Cancel() est effectué. Cela signalera l'annulation. Ensuite, la `sumPageSizesTask` variable est assignée à partir de la `SumPageSizesAsync` méthode. Les deux tâches sont ensuite transmises à Task.WhenAny(Task[]) , qui se poursuivra quand l'une des deux tâches sera terminée.

Créer la méthode de taille de page Sum asynchrone

Sous la `Main` méthode, ajoutez la `SumPageSizesAsync` méthode :

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

La méthode commence par instancier et démarrer un Stopwatch . Il parcourt ensuite chaque URL dans le `s_urlList` et appelle `ProcessUrlAsync` . Avec chaque itération, le `s_cts.Token` est passé dans la `ProcessUrlAsync` méthode et le code retourne un Task<TResult> , où `TResult` est un entier :

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Ajouter une méthode de traitement

Ajoutez la méthode suivante en `ProcessUrlAsync` dessous de la `SumPageSizesAsync` méthode :

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync();
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Pour une URL donnée, la méthode utilise l' `client` instance fournie pour obtenir la réponse en tant que `byte[]` . L' CancellationToken instance est passée dans les HttpClient.GetAsync(String, CancellationToken) HttpContent.ReadAsByteArrayAsync() méthodes et. Le `token` est utilisé pour s'inscrire à l'annulation demandée. La longueur est retournée une fois que l'URL et la longueur sont écrites dans la console.

Exemple de sortie d'application

```
Application started.
Press the ENTER key to cancel...

https://docs.microsoft.com                                      37,357
https://docs.microsoft.com/aspnet/core                          85,589
https://docs.microsoft.com/azure                               398,939
https://docs.microsoft.com/azure/devops                         73,663
https://docs.microsoft.com/dotnet                               67,452
https://docs.microsoft.com/dynamics365                          48,582
https://docs.microsoft.com/education                            22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* pour l'exemple.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");
```

```
        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }

            Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync();
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Voir aussi

- CancellationToken
- CancellationTokenSource
- Programmation asynchrone avec Async et await (C#)

Étapes suivantes

Annuler des tâches async après une période spécifique (C#)

Annuler des tâches async après une période spécifique (C#)

02/11/2020 • 3 minutes to read • Edit Online

Vous pouvez annuler une opération asynchrone au bout d'un certain temps à l'aide de la CancellationTokenSource.CancelAfter méthode si vous ne souhaitez pas attendre la fin de l'opération. Cette méthode planifie l'annulation de toutes les tâches associées qui ne sont pas terminées dans le délai indiqué par l' `CancelAfter` expression.

Cet exemple ajoute au code développé dans annuler une liste de tâches (C#) pour télécharger une liste de sites Web et afficher la longueur du contenu de chacun d'eux.

Ce didacticiel contient les sections suivantes :

- Mise à jour d'une application console .NET existante
- Planification d'une annulation

Prérequis

Ce didacticiel requiert les éléments suivants :

- Vous êtes censé avoir créé une application dans le didacticiel annuler une liste de tâches (C#)
- .NET 5,0 ou kit de développement logiciel (SDK) ultérieur
- Environnement de développement intégré (IDE)
  - Nous vous recommandons Visual Studio, Visual Studio Code ou Visual Studio pour Mac

Mettre à jour le point d'entrée de l'application

Remplacez la `Main` méthode existante par le code suivant :

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }

    Console.WriteLine("Application ending.");
}
```

La méthode mise à jour `Main` écrit quelques messages d'instructions dans la console. Dans le bloc try, un appel à CancellationTokenSource.CancelAfter(Int32) pour planifier une annulation. Cela signalera l'annulation après un certain laps de temps.

Ensuite, la `SumPageSizesAsync` méthode est attendue. Si le traitement de toutes les URL se produit plus rapidement

que l'annulation planifiée, l'application se termine. Toutefois, si l'annulation planifiée est déclenchée avant le traitement de toutes les URL, une TaskCanceledException est levée.

Exemple de sortie d'application

```
Application started.

https://docs.microsoft.com                                        37,357
https://docs.microsoft.com/aspnet/core                            85,589
https://docs.microsoft.com/azure                                 398,939
https://docs.microsoft.com/azure/devops                           73,663

Tasks cancelled: timed out.

Application ending.
```

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* pour l'exemple.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
```

```
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Voir aussi

- CancellationToken
- CancellationTokenSource
- Programmation asynchrone avec Async et await (C#)
- Annuler une liste de tâches (C#)

Traiter les tâches asynchrones à mesure qu'elles se terminent (C#)

02/11/2020 • 7 minutes to read • Edit Online

À l'aide de Task.WhenAny , vous pouvez démarrer plusieurs tâches en même temps et les traiter une par une lorsqu'elles sont terminées, au lieu de les traiter dans l'ordre dans lequel elles sont démarrées.

L'exemple suivant utilise une requête pour créer une collection de tâches. Chaque tâche télécharge le contenu d'un site web spécifié. À chaque itération d'une boucle while, un appel attendu à WhenAny retourne la tâche de la collection de tâches dont le téléchargement se termine en premier. Cette tâche est supprimée de la collection et traitée. La boucle se répète jusqu'à ce que la collection ne contienne plus aucune tâche.

Créer un exemple d'application

Créez une application de console .NET Core. Vous pouvez en créer un à l'aide de la commande dotnet New console ou de Visual Studio. Ouvrez le fichier *Program.cs* dans votre éditeur de code favori.

Remplacer les instructions using

Remplacez les instructions using existantes par les déclarations suivantes :

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

Ajouter des champs

Dans la `Program` définition de classe, ajoutez les deux champs suivants :

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

Le `HttpClient` expose la possibilité d'envoyer des requêtes http et de recevoir des réponses http. Le `s_urlList` contient toutes les URL que l'application envisage de traiter.

Mettre à jour le point d'entrée de l'application

Le point d'entrée principal dans l'application console est la `Main` méthode. Remplacez la méthode existante par le code suivant :

```
static Task Main() => SumPageSizesAsync();
```

La méthode mise à jour `Main` est maintenant considérée comme une main asynchrone, ce qui permet d'obtenir un point d'entrée asynchrone dans l'exécutable. Il s'agit d'un appel à `SumPageSizesAsync` .

Créer la méthode de taille de page Sum asynchrone

Sous la `Main` méthode, ajoutez la `SumPageSizesAsync` méthode :

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

La méthode commence par instancier et démarrer un Stopwatch . Il comprend ensuite une requête qui, lorsqu'elle est exécutée, crée une collection de tâches. Chaque appel à `ProcessUrlAsync` dans le code suivant retourne un Task<TResult>, où `TResult` est un entier :

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

En raison d'une exécution différée avec LINQ, vous appelez Enumerable.ToList pour démarrer chaque tâche.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

La `while` boucle effectue les étapes suivantes pour chaque tâche de la collection :

1. Attend un appel à `WhenAny` pour identifier la première tâche de la collection dont le téléchargement est terminé.

   ```
   Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Elle supprime cette tâche de la collection.

   ```
   downloadTasks.Remove(firstFinishedTask);
   ```

3. Elle attend `finishedTask`, qui est retourné par un appel à `ProcessUrlAsync`. La variable `finishedTask` est un Task<TResult> où `TResult` est un entier. La tâche est déjà terminée, mais vous l'attendez pour récupérer la longueur du site web téléchargé, comme le montre l'exemple suivant. Si la tâche est défaillante, `await` lèvera la première exception enfant stockée dans le `AggregateException` , contrairement à la lecture de la Task<TResult>.Result propriété, qui lèverait `AggregateException` .

   ```
   total += await finishedTask;
   ```

Ajouter une méthode de traitement

Ajoutez la méthode suivante en `ProcessUrlAsync` dessous de la `SumPageSizesAsync` méthode :

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Pour une URL donnée, la méthode utilise l' `client` instance fournie pour obtenir la réponse en tant que `byte[]` . La longueur est retournée une fois que l'URL et la longueur sont écrites dans la console.

Exécutez le programme plusieurs fois pour vérifier que les longueurs téléchargées n'apparaissent pas toujours dans le même ordre.

Caution

Vous pouvez utiliser `WhenAny` dans une boucle, comme décrit dans l'exemple, pour résoudre les problèmes qui impliquent un petit nombre de tâches. Cependant, d'autres approches sont plus efficaces si vous avez un grand nombre de tâches à traiter. Pour plus d'informations et d'exemples, consultez traitement des tâches lorsqu'elles sont terminées.

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* pour l'exemple.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
```

```
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
        };

        static Task Main() => SumPageSizesAsync();

        static async Task SumPageSizesAsync()
        {
            var stopwatch = Stopwatch.StartNew();

            IEnumerable<Task<int>> downloadTasksQuery =
                from url in s_urlList
                select ProcessUrlAsync(url, s_client);

            List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

            int total = 0;
            while (downloadTasks.Any())
            {
                Task<int> finishedTask = await Task.WhenAny(downloadTasks);
                downloadTasks.Remove(finishedTask);
                total += await finishedTask;
            }

            stopwatch.Stop();

            Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
            Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
        }

        static async Task<int> ProcessUrlAsync(string url, HttpClient client)
        {
            byte[] content = await client.GetByteArrayAsync(url);
            Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

            return content.Length;
        }
    }
}
// Example output:
// https://docs.microsoft.com/windows                                25,513
// https://docs.microsoft.com/gaming                                 30,705
// https://docs.microsoft.com/dotnet                                 69,626
// https://docs.microsoft.com/dynamics365                            50,756
// https://docs.microsoft.com/surface                                35,519
// https://docs.microsoft.com                                        39,531
// https://docs.microsoft.com/azure/devops                           75,837
// https://docs.microsoft.com/xamarin                                60,284
// https://docs.microsoft.com/system-center                          43,444
// https://docs.microsoft.com/enterprise-mobility-security           28,946
// https://docs.microsoft.com/microsoft-365                          43,278
// https://docs.microsoft.com/visualstudio                           31,414
// https://docs.microsoft.com/office                                 42,292
// https://docs.microsoft.com/azure                                 401,113
// https://docs.microsoft.com/graph                                  46,831
// https://docs.microsoft.com/education                              25,098
// https://docs.microsoft.com/powershell                             58,173
// https://docs.microsoft.com/aspnet/core                            87,763
// https://docs.microsoft.com/sql                                    53,362

// Total bytes returned: 1,249,485
// Elapsed time:          00:00:02.7068725
```

Voir aussi

- WhenAny

- Programmation asynchrone avec Async et await (C#)

Accès asynchrone aux fichiers (C#)

02/11/2020 • 9 minutes to read • Edit Online

Vous pouvez utiliser la fonctionnalité Async pour accéder à des fichiers. La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels ni fractionner votre code entre plusieurs méthodes ou expressions lambda. Pour rendre le code synchrone asynchrone, il vous suffit d'appeler une méthode asynchrone au lieu d'une méthode synchrone, puis d'ajouter quelques mots clés au code.

Vous pouvez considérer les raisons suivantes pour ajouter de l'asynchrone aux appels d'accès aux fichiers :

- L'asynchronisme rend les applications d'interface utilisateur plus réactives parce que le thread d'interface utilisateur qui lance l'exécution peut effectuer d'autres tâches. Si le thread d'interface utilisateur doit exécuter du code qui prend du temps (par exemple, plus de 50 millisecondes), l'interface utilisateur peut figer jusqu'à ce que l'E/S soit terminée et que le thread d'interface utilisateur puisse traiter à nouveau des entrées au clavier et à la souris et d'autres événements.
- L'asynchronisme améliore l'extensibilité d'ASP.NET et d'autres applications serveur en réduisant le besoin de threads. Si l'application utilise un thread dédié par réponse et que mille demandes sont traitées simultanément, mille threads sont nécessaires. Les opérations asynchrones n'ont souvent pas besoin d'utiliser un thread pendant l'attente. Elles utilisent le thread de terminaison d'E/S existant brièvement à la fin.
- La latence d'une opération d'accès à un fichier peut être très basse sous les conditions actuelles, mais la latence peut considérablement augmenter à l'avenir. Par exemple, un fichier peut être déplacé vers un serveur qui se trouve à travers le monde.
- La charge mémoire supplémentaire pour l'utilisation de la fonctionnalité Async est faible.
- Les tâches asynchrones peuvent facilement être exécutées en parallèle.

Utiliser les classes appropriées

Les exemples simples de cette rubrique illustrent File.WriteAllTextAsync et File.ReadAllTextAsync . Pour un contrôle fini sur les opérations d'e/s de fichier, utilisez la FileStream classe, qui a une option qui provoque l'exécution d'e/s asynchrones au niveau du système d'exploitation. En utilisant cette option, vous pouvez éviter de bloquer un thread de pool de threads dans de nombreux cas. Pour l'activer, spécifiez l'argument `useAsync=true` ou `options=FileOptions.Asynchronous` dans l'appel de constructeur.

Vous ne pouvez pas utiliser cette option avec StreamReader et StreamWriter si vous les ouvrez directement en spécifiant un chemin d'accès de fichier. Toutefois, vous pouvez utiliser cette option si vous leur fournissez un Stream ouvert par la classe FileStream. Les appels asynchrones sont plus rapides dans les applications d'interface utilisateur même si un thread de pool de threads est bloqué, car le thread d'interface utilisateur n'est pas bloqué pendant l'attente.

Écrire du texte

Les exemples suivants écrivent du texte dans un fichier. À chaque instruction await, la méthode s'arrête immédiatement. Quand l'E/S de fichier est terminée, la méthode reprend à l'instruction qui suit l'instruction await. Le modificateur Async est dans la définition des méthodes qui utilisent l'instruction await.

Exemple simple

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Exemple de contrôle fini

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

L'exemple d'origine comprend l'instruction `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, qui est une contraction des deux instructions suivantes :

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

La première instruction retourne une tâche et provoque le début du traitement du fichier. La deuxième instruction avec await provoque la fin immédiate de la méthode et retourne une tâche différente. Quand le traitement du fichier se termine plus loin, l'exécution retourne à l'instruction qui suit l'attente.

Lire le texte

Les exemples suivants lisent du texte à partir d'un fichier.

Exemple simple

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Exemple de contrôle fini

Le texte est mis en mémoire tampon et, dans cet exemple, est placé dans un StringBuilder. Contrairement à l'exemple précédent, l'évaluation de l'instruction await génère une valeur. La ReadAsync méthode retourne un Task

<Int32>, donc l'évaluation de l'expression await produit une `Int32` valeur `numRead` une fois l'opération terminée. Pour plus d'informations, consultez Types de retour Async (C#).

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

E/s asynchrones parallèles

Les exemples suivants illustrent le traitement en parallèle en écrivant 10 fichiers texte.

Exemple simple

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Exemple de contrôle fini

Pour chaque fichier, la méthode WriteAsync retourne une tâche qui est ensuite ajoutée à une liste des tâches. L'instruction `await Task.WhenAll(tasks);` quitte la méthode et reprend dans cette dernière quand le traitement du fichier est terminé pour toutes les tâches.

L'exemple ferme toutes les instances de FileStream dans un bloc `finally` une fois les tâches effectuées. Si chaque `FileStream` était plutôt créé dans une instruction `using`, `FileStream` pourrait être libéré avant que la tâche ne soit terminée.

Toute amélioration des performances est presque entièrement issue du traitement parallèle et non du traitement asynchrone. L'asynchronie présente l'avantage de ne pas bloquer plusieurs threads et de ne pas bloquer le thread d'interface utilisateur.

```
public async Task ProcessMulitpleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

Quand vous utilisez les méthodes WriteAsync et ReadAsync, vous pouvez spécifier un CancellationToken, qui vous permet d'annuler l'opération en cours de route. Pour plus d'informations, consultez Annulation dans les threads managés.

Voir aussi

- Programmation asynchrone avec Async et await (C#)
- Types de retour Async (C#)

Attributs (C#)

02/11/2020 • 9 minutes to read • Edit Online

Les attributs fournissent une méthode puissante permettant d'associer des métadonnées ou des informations déclaratives avec du code (assemblys, types, méthodes, propriétés, etc.). Une fois associé à une entité de programme, l'attribut peut être interrogé à l'exécution à l'aide d'une technique appelée *réflexion*. Pour plus d'informations, consultez Réflexion (C#).

Les attributs ont les propriétés suivantes :

- Les attributs ajoutent des métadonnées à un programme. Les *métadonnées* sont des informations sur les types définis dans un programme. Tous les assemblys .NET contiennent un ensemble spécifié de métadonnées qui décrivent les types et membres de types définis dans l'assembly. Vous pouvez ajouter des attributs personnalisés pour spécifier des informations supplémentaires si nécessaire. Pour plus d'informations, consultez Création d'attributs personnalisés (C#).
- Vous pouvez appliquer un ou plusieurs attributs à des modules ou des assemblys entiers ou à de plus petits éléments de programmes, comme des classes et des propriétés.
- Les attributs peuvent accepter des arguments de la même façon que les méthodes et les propriétés.
- Votre programme peut examiner ses propres métadonnées ou celles d'autres programmes grâce à la réflexion. Pour plus d'informations, consultez Accès à des attributs à l'aide de la réflexion (C#).

Utilisation des attributs

Les attributs peuvent être placés sur la quasi-totalité des déclarations, même si un attribut donné peut restreindre les types de déclarations sur lesquels il est valide. En C#, vous spécifiez un attribut en plaçant son nom entre crochets ([]) au-dessus de la déclaration de l'entité à laquelle il s'applique.

Dans cet exemple, l'attribut SerializableAttribute est utilisé pour appliquer une caractéristique spécifique à une classe :

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Une méthode avec l'attribut DllImportAttribute est déclarée comme dans l'exemple suivant :

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

Plusieurs attributs peuvent être placés dans une déclaration comme dans l'exemple suivant :

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Certains attributs peuvent être spécifiés plusieurs fois pour une entité donnée. ConditionalAttribute est un exemple d'attribut à utilisation multiple :

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **NOTE**
> Par convention, tous les noms d'attributs se terminent par le mot « Attribute » pour les différencier d'autres éléments dans les bibliothèques .NET. Toutefois, il est inutile de spécifier le suffixe d'attribut lorsque les attributs sont utilisés dans le code. Par exemple, `[DllImport]` est équivalent à `[DllImportAttribute]` , mais `DllImportAttribute` est le nom réel de l'attribut dans la bibliothèque de classes .net.

Paramètres d'attributs

Beaucoup d'attributs possèdent des paramètres. Ceux-ci peuvent être positionnels, sans nom ou nommés. Les paramètres positionnels doivent être spécifiés dans un certain ordre et ne peut pas être omis. Les paramètres nommés sont facultatifs et peuvent être spécifiés dans n'importe quel ordre. Les paramètres positionnels sont spécifiés en premier. Par exemple, ces trois attributs sont équivalents :

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

Le premier paramètre, le nom de la DLL, est positionnel et se place toujours en premier ; les autres sont nommés. Dans ce cas, les deux paramètres nommés ont la valeur false par défaut ; ainsi, ils peuvent être omis. Les paramètres positionnels correspondent aux paramètres du constructeur d'attribut. Les paramètres nommés ou facultatifs correspondent aux propriétés ou champs de l'attribut. Reportez-vous à la documentation de chaque attribut pour plus d'informations sur les valeurs des paramètres par défaut.

Cibles d'attribut

La *cible* d'un attribut est l'entité à laquelle s'applique l'attribut. Par exemple, un attribut peut s'appliquer à une classe, à une méthode particulière ou à un assembly entier. Par défaut, un attribut s'applique à l'élément qui le suit. Mais vous pouvez également identifier de manière explicite, par exemple, si un attribut s'applique à une méthode, à son paramètre ou à sa valeur renvoyée.

Pour identifier de manière explicite une cible d'attribut, utilisez la syntaxe suivante :

```
[target : attribute-list]
```

La liste des valeurs `target` possibles est présentée dans le tableau suivant.

VALEUR CIBLE	S'APPLIQUE À
`assembly`	Assembly entier
`module`	Module d'assembly actuel
`field`	Champ dans une classe ou un struct

VALEUR CIBLE	S'APPLIQUE À
`event`	Événement
`method`	Méthode ou accesseurs de propriété `get` et `set`
`param`	Paramètres de méthode ou paramètres d'accesseur de propriété `set`
`property`	Propriété
`return`	Valeur de retour d'une méthode, indexeur de propriété ou accesseur de propriété `get`
`type`	Struct, classe, interface, énumération ou délégué

Vous devez spécifier la valeur cible de `field` pour appliquer un attribut au champ de stockage créé pour une propriété implémentée automatiquement.

L'exemple suivant montre comment appliquer des attributs à des modules et assemblys. Pour plus d'informations, consultez Attributs courants (C#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

L'exemple suivant montre comment appliquer des attributs à des méthodes, des paramètres de méthode et des valeurs de retour de méthode en C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to return value
[return: ValidatedContract]
int Method3() { return 0; }
```

NOTE

Quelle que soit la cible sur laquelle `ValidatedContract` est défini comme étant valide, la cible `return` doit être spécifiée, même si `ValidatedContract` a été défini pour s'appliquer seulement aux valeurs de retour. En d'autres termes, le compilateur n'utilise pas les informations `AttributeUsage` pour résoudre les cibles d'attribut ambiguës. Pour plus d'informations, consultez AttributeUsage (C#).

Utilisations courantes des attributs

La liste suivante comprend certaines des utilisations courantes des attributs dans le code :

- Marquer des méthodes avec l'attribut `WebMethod` dans les services web pour indiquer que la méthode doit

pouvoir être appelée via le protocole SOAP. Pour plus d'informations, consultez WebMethodAttribute.

- Décrire comment marshaler les paramètres de méthode en cas d'interaction avec du code natif. Pour plus d'informations, consultez MarshalAsAttribute.
- Décrire les propriétés COM des classes, méthodes et interfaces.
- Appeler du code non managé à l'aide de la classe DllImportAttribute.
- Décrire un assembly : titre, version, description ou marque.
- Décrire les membres d'une classe à sérialiser à des fins de persistance.
- Décrire le mappage entre les membres de classe et des nœuds XML à des fins de sérialisation XML.
- Décrire les exigences de sécurité des méthodes.
- Spécifier les caractéristiques employées pour appliquer la sécurité.
- Contrôler les optimisations par le compilateur juste-à-temps (JIT) pour que le code reste facile à déboguer.
- Obtenir des informations sur l'appelant d'une méthode.

Sections connexes

Pour plus d'informations, voir :

- Création d'attributs personnalisés (C#)
- Accès à des attributs à l'aide de la réflexion (C#)
- Guide pratique pour créer une Union C/C++ à l'aide d'attributs (C#)
- Attributs courants (C#)
- Informations relatives à l'appelant (C#)

Voir aussi

- Guide de programmation C#
- Réflexion (C#)
- Attributs
- Utilisation d'attributs en C #

Création d'attributs personnalisés (C#)

02/11/2020 • 2 minutes to read • Edit Online

Vous pouvez créer vos propres attributs personnalisés en définissant une classe d'attributs. Cette classe dérive directement ou indirectement d'Attribute, ce qui permet d'identifier rapidement et facilement des définitions d'attributs dans des métadonnées. Supposons que vous souhaitiez baliser des types avec le nom du programmeur qui les a écrits. Vous pouvez définir une classe d'attributs `Author` personnalisés :

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

Le nom de la classe `AuthorAttribute` est le nom de l'attribut, `Author` , et le `Attribute` suffixe. Étant dérivée de `System.Attribute` , il s'agit donc d'une classe d'attributs personnalisés. Les paramètres du constructeur sont les paramètres positionnels de l'attribut personnalisé. Dans cet exemple, `name` est un paramètre positionnel. Les propriétés ou champs de type public accessibles en lecture/écriture sont des paramètres nommés. Dans le cas présent, `version` est le seul paramètre nommé. Notez l'utilisation de l'attribut `AttributeUsage` pour rendre l'attribut `Author` valide uniquement sur les déclarations de classe et de `struct` .

Vous pouvez utiliser ce nouvel attribut de la manière suivante :

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` a un paramètre nommé, `AllowMultiple` , qui permet de déclarer un attribut personnalisé comme étant à usage unique ou multiple. Dans l'exemple de code suivant, un attribut à usage multiple est créé.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // multiuse attribute
]
public class AuthorAttribute : System.Attribute
```

Dans l'exemple de code suivant, plusieurs attributs du même type sont appliqués à une classe.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

Voir aussi

- System.Reflection
- Guide de programmation C#
- Écriture des attributs personnalisés
- Réflexion (C#)
- Attributs (C#)
- Accès à des attributs à l'aide de la réflexion (C#)
- AttributeUsage (C#)

Accès à des attributs à l'aide de la réflexion (C#)

02/11/2020 • 3 minutes to read • Edit Online

La définition d'attributs personnalisés et leur ajout à votre code source présentent peu d'intérêt si vous ne pouvez pas ensuite récupérer et manipuler ces informations. La réflexion vous permet de récupérer les informations qui ont été définies à l'aide d'attributs personnalisés. La méthode clé est `GetCustomAttributes`. Elle retourne un tableau d'objets qui sont les équivalents des attributs du code source au moment de l'exécution. Cette méthode a plusieurs versions surchargées. Pour plus d'informations, consultez Attribute.

Par exemple, cette spécification d'attribut :

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

est semblable à celle-ci d'un point de vue conceptuel :

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.version = 1.1;
```

Toutefois, le code n'est pas exécuté tant qu'une requête n'est pas effectuée sur `SampleClass` pour obtenir les attributs. L'appel de `GetCustomAttributes` sur `SampleClass` entraîne la construction et l'initialisation d'un objet `Author` comme illustré ci-dessus. Si la classe a d'autres attributs, les autres objets attribut sont construits de la même façon. `GetCustomAttributes` retourne ensuite l'objet `Author` et les autres objets attribut dans un tableau. Vous pouvez ensuite itérer sur ce tableau, déterminer quels attributs ont été appliqués en fonction du type de chaque élément du tableau et extraire des informations des objets attribut.

Exemple

Voici un exemple complet. Un attribut personnalisé est défini, appliqué à plusieurs entités, puis récupéré par réflexion.

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return name;
    }
}
```

```
// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("   {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

Voir aussi

- System.Reflection
- Attribute
- Guide de programmation C#
- Récupération des informations stockées dans les attributs
- Réflexion (C#)

- Attributs (C#)
- Création d'attributs personnalisés (C#)

Guide pratique pour créer une Union C/C++ à l'aide d'attributs (C#)

02/11/2020 • 2 minutes to read • Edit Online

À l'aide d'attributs, vous pouvez personnaliser la disposition des structs en mémoire. Par exemple, vous pouvez créer ce qu'on appelle une union en C/C++ à l'aide des attributs `StructLayout(LayoutKind.Explicit)` et `FieldOffset`.

Exemple

Dans ce segment de code, tous les champs de `TestUnion` débutent au même emplacement en mémoire.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

Exemple

Voici un autre exemple où les champs débutent à différents emplacements définis explicitement.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Les deux champs entiers, `i1` et `i2`, partagent leurs emplacements de mémoire avec `lg`. Ce type de contrôle sur la disposition des structs est utile quand vous utilisez des appels de code non managé.

Voir aussi

- System.Reflection
- Attribute
- Guide de programmation C#
- Attributs
- Réflexion (C#)
- Attributs (C#)
- Création d'attributs personnalisés (C#)
- Accès à des attributs à l'aide de la réflexion (C#)

Collections (C#)

02/11/2020 • 21 minutes to read • Edit Online

Pour de nombreuses applications, vous voulez créer et gérer des groupes d'objets connexes. Il existe deux manières de grouper des objets : en créant des tableaux d'objets ou des collections d'objets.

Les tableaux s'avèrent particulièrement utiles pour créer et utiliser un nombre fixe d'objets fortement typés. Pour plus d'informations sur les tableaux, consultez Tableaux.

Les collections offrent plus de souplesse quand il s'agit d'utiliser des groupes d'objets. Contrairement aux tableaux, le groupe d'objets que vous utilisez peut être développé et réduit de manière dynamique selon les modifications de l'application. Pour certaines collections, vous pouvez assigner une clé à un objet que vous placez dans la collection pour vous permettre de récupérer rapidement l'objet à l'aide de la clé.

Une collection est une classe, vous devez déclarer une instance de la classe avant de pouvoir ajouter des éléments à la collection.

Si votre collection contient des éléments d'un seul type de données, vous pouvez utiliser une des classes dans l'espace de noms System.Collections.Generic. Une collection générique applique la cohérence des types pour éviter qu'un autre type puisse y être ajouté. Quand vous récupérez un élément d'une collection générique, il n'est pas utile de déterminer son type de données ou de le convertir.

> **NOTE**
>
> Pour les exemples de cette rubrique, ajoutez des instructions using pour les espaces de noms `System.Collections.Generic` et `System.Linq`.

Dans cette rubrique

- Utilisation d'une collection simple
- Types de collections
  - Classes System.Collections.Generic
  - Classes System.Collections.Concurrent
  - Classes System.Collections
- Implémentation d'une collection de paires clé-valeur
- Utilisation de LINQ pour accéder à une collection
- Tri d'une collection
- Définition d'une collection personnalisée
- Iterators

Utilisation d'une collection simple

Les exemples de cette section utilisent la classe List<T> générique, qui vous permet d'utiliser une liste d'objets fortement typée.

L'exemple suivant crée une liste de chaînes, puis itère au sein des chaînes à l'aide d'une instruction foreach.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Si le contenu d'une collection est connu d'avance, vous pouvez utiliser un *initialiseur de collection* pour initialiser la collection. Pour plus d'informations, consultez la page Initialiseurs d'objets et de collections.

L'exemple suivant est identique à l'exemple précédent, à la différence qu'un initialiseur de collection est utilisé pour ajouter des éléments à la collection.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Vous pouvez utiliser une instruction for au lieu d'une instruction `foreach` pour itérer au sein d'une collection. Pour cela, accédez aux éléments de la collection à la position d'index. L'index des éléments commence à 0 et se termine au nombre d'éléments moins 1.

L'exemple suivant itère au sein des éléments d'une collection à l'aide de `for` au lieu de `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

L'exemple suivant supprime un élément de la collection en spécifiant l'objet à supprimer.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

L'exemple suivant supprime les éléments d'une liste générique. À la place d'une instruction `foreach`, une instruction for, qui itère dans l'ordre décroissant, est utilisée. En effet, avec la méthode RemoveAt, les éléments après l'élément supprimé ont une valeur d'index moins élevée.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Pour le type d'éléments de List<T>, vous pouvez également définir votre propre classe. Dans l'exemple suivant, la classe `Galaxy` qui est utilisée par List<T> est définie dans le code.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Types de collections

De nombreuses collections courantes sont fournies par .NET. Chaque type de collection est conçu dans un but spécifique.

Certaines des classes de collection courantes sont décrites dans cette section :

- Classes System.Collections.Generic
- Classes System.Collections.Concurrent
- Classes System.Collections

Classes System.Collections.Generic

Vous pouvez créer une collection générique en utilisant l'une des classes dans l'espace de noms System.Collections.Generic. Une collection générique est utile quand chaque élément de la collection a le même type de données. Une collection générique applique un typage fort en autorisant uniquement l'ajout des types de données souhaités.

Le tableau suivant liste quelques classes de l'espace de noms System.Collections.Generic fréquemment utilisées :

CLASSE	DESCRIPTION
Dictionary<TKey,TValue>	Représente une collection de paires clé/valeur organisées en fonction de la clé.
List<T>	Représente une liste d'objets accessibles par index. Fournit des méthodes de recherche, de tri et de modification de listes.
Queue<T>	Représente une collection d'objets premier entré, premier sorti (FIFO).

CLASSE	DESCRIPTION
SortedList<TKey,TValue>	Représente une collection de paires clé/valeur triées par clé en fonction de l'implémentation IComparer<T> associée.
Stack<T>	Représente une collection d'objets dernier entré, premier sorti (LIFO).

Pour plus d'informations, consultez Types de collections couramment utilisés, Sélection d'une classe de collection et System.Collections.Generic.

Classes System.Collections.Concurrent

Dans .NET Framework 4 et versions ultérieures, les collections de l' System.Collections.Concurrent espace de noms fournissent des opérations thread-safe efficaces pour accéder aux éléments de collection à partir de plusieurs threads.

Les classes de l'espace de noms System.Collections.Concurrent doivent être utilisées à la place des types correspondants dans les espaces de noms System.Collections.Generic et System.Collections chaque fois que plusieurs threads accèdent simultanément à la collection. Pour plus d'informations, consultez Collections thread-safe et System.Collections.Concurrent.

Certaines classes incluses dans l'espace de noms System.Collections.Concurrent sont BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T> et ConcurrentStack<T>.

Classes System.Collections

Les classes de l'espace de noms System.Collections ne stockent pas les éléments comme des objets spécifiquement typés, mais comme des objets de type `Object`.

Si possible, vous devez utiliser les collections génériques dans l'espace de noms System.Collections.Generic ou System.Collections.Concurrent à la place des types hérités de l'espace de noms `System.Collections`.

Le tableau suivant répertorie certaines des classes fréquemment utilisées de l'espace de noms `System.Collections` :

CLASSE	DESCRIPTION
ArrayList	Représente un tableau d'objets dont la taille est augmentée de manière dynamique selon les besoins.
Hashtable	Représente une collection de paires clé/valeur qui sont organisées en fonction du code de hachage de la clé.
Queue	Représente une collection d'objets premier entré, premier sorti (FIFO).
Stack	Représente une collection d'objets dernier entré, premier sorti (LIFO).

L'espace de noms System.Collections.Specialized fournit des classes de collection spécialisées et fortement typées, telles que les collections à chaîne unique et les dictionnaires de liste liée et hybrides.

Implémentation d'une collection de paires clé/valeur

La collection générique Dictionary<TKey,TValue> vous permet d'accéder aux éléments d'une collection à l'aide de la clé de chaque élément. Chaque ajout au dictionnaire se compose d'une valeur et de sa clé associée. La récupération d'une valeur à l'aide de sa clé est très rapide, car la classe `Dictionary` est implémentée en tant que

table de hachage.

L'exemple suivant crée une collection `Dictionary` et itère au sein du dictionnaire à l'aide d'une instruction `foreach` .

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Au lieu d'utiliser un initialiseur de collection pour générer la collection `Dictionary`, vous pouvez remplacer les méthodes `BuildDictionary` et `AddToDictionary` par la méthode suivante.

```
private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

L'exemple suivant utilise la méthode ContainsKey et la propriété Item[] de `Dictionary` pour rechercher rapidement un élément par clé. En C#, la propriété `Item` vous permet d'accéder à un élément de la collection `elements` à l'aide d' `elements[symbol]`.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

L'exemple suivant utilise à la place la méthode TryGetValue pour rechercher rapidement un élément par clé.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Utilisation de LINQ pour accéder à une collection

LINQ (Language-Integrated Query) peut être utilisé pour accéder aux collections. Les requêtes LINQ fournissent des fonctionnalités de filtrage, de classement et de regroupement. Pour plus d'informations, consultez prise en main avec LINQ en C#.

L'exemple suivant exécute une requête LINQ sur un `List` générique. La requête LINQ retourne une autre collection qui contient les résultats.

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Tri d'une collection

L'exemple suivant illustre une procédure de tri d'une collection. L'exemple trie les instances de la classe `Car` stockées dans un List<T>. La classe `Car` implémente l'interface IComparable<T>, ce qui implique l'implémentation de la méthode CompareTo.

Chaque appel à la méthode CompareTo effectue une comparaison unique qui est utilisée pour le tri. Le code écrit par l'utilisateur dans la méthode `CompareTo` retourne une valeur pour chaque comparaison de l'objet actuel avec un autre objet. La valeur retournée est inférieure à zéro si l'objet actuel est inférieur à l'autre objet, supérieure à zéro l'objet actuel est supérieur à l'autre et égale à zéro s'ils sont égaux. Cela vous permet de définir dans le code les critères définissant « supérieur à », « inférieur à » et « égal à ».

Dans la méthode `ListCars`, l'instruction `cars.Sort()` trie la liste. Cet appel à la méthode Sort de List<T> entraîne l'appel automatique de la méthode `CompareTo` pour les objets `Car` dans `List`.

```
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
```

```
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
    //  red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

Définition d'une collection personnalisée

Vous pouvez définir une collection en implémentant l'interface IEnumerable<T> ou IEnumerable.

Bien que vous puissiez définir une collection personnalisée, il est généralement préférable d'utiliser les collections incluses dans .NET, qui sont décrites dans genres de collections , plus haut dans cet article.

L'exemple suivant définit une classe de collection personnalisée nommée `AllColors`. Cette classe implémente l'interface IEnumerable, ce qui implique l'implémentation de la méthode GetEnumerator.

La méthode `GetEnumerator` retourne une instance de la classe `ColorEnumerator`. `ColorEnumerator` implémente l'interface IEnumerator, ce qui implique l'implémentation de la propriété Current, la méthode MoveNext et la méthode Reset.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
```

```
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}
```

Iterators

Un *itérateur* est utilisé pour exécuter une itération personnalisée sur une collection. Un itérateur peut être une méthode ou un accesseur `get`. Un itérateur utilise une instruction yield return pour retourner chaque élément de la collection un par un.

Vous appelez un itérateur en utilisant une instruction foreach. Chaque itération de la boucle `foreach` appelle l'itérateur. Quand une instruction `yield return` est atteinte dans l'itérateur, une expression est retournée et la localisation actuelle dans le code est retenue. L'exécution est redémarrée à partir de cet emplacement la prochaine fois que l'itérateur est appelé.

Pour plus d'informations, consultez Itérateurs (C#).

L'exemple suivant utilise une méthode Iterator. La méthode Iterator a une instruction `yield return` qui se trouve à l'intérieur d'une boucle for. Dans la méthode `ListEvenNumbers`, chaque itération du corps de l'instruction `foreach` crée un appel à la méthode Iterator, qui continue sur l'instruction `yield return` suivante.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Voir aussi

- Initialiseurs d'objets et de collections
- Concepts de programmation (C#)
- Option Strict Statement
- LINQ to Objects (C#)
- Parallel LINQ (PLINQ)
- Collections et structures de données
- Sélection d'une classe de collection

- Comparaisons et tris dans les collections
- Quand utiliser des collections génériques

Covariance et contravariance (C#)

02/11/2020 • 5 minutes to read • Edit Online

En C#, la covariance et la contravariance permettent la conversion de références implicite pour les types tableau, les types délégués et les arguments de type générique. La covariance conserve la compatibilité d'assignation et la contravariance l'inverse.

Le code suivant illustre la différence entre la compatibilité d'assignation, la covariance et la contravariance.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

La covariance pour les tableaux permet la conversion implicite d'un tableau d'un type plus dérivé en un tableau d'un type moins dérivé. Cette opération n'est cependant pas sécurisée au niveau des types, comme le montre l'exemple de code suivant.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

La prise en charge de la covariance et la contravariance pour les groupes de méthodes permet la correspondance des signatures de méthode avec des types délégués. Ceci vous permet d'affecter aux délégués non seulement les méthodes ayant des signatures correspondantes, mais aussi des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué. Pour plus d'informations, consultez Variance dans les délégués (C#) et Utilisation de la variance dans les délégués (C#).

L'exemple de code suivant montre la prise en charge de la covariance et de la contravariance pour les groupes de méthode.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

Dans .NET Framework 4 et versions ultérieures, C# prend en charge la covariance et la contravariance dans les interfaces et les délégués génériques, et permet la conversion implicite de paramètres de type générique. Pour plus d'informations, consultez Variance dans les interfaces génériques (C#) et Variance dans les délégués (C#).

L'exemple de code suivant illustre la conversion de références implicite pour les interfaces génériques.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Une interface ou un délégué générique est dit de type *variant* si ses paramètres génériques sont déclarés covariants ou contravariants. C# vous permet de créer vos propres interfaces et délégués de type variant. Pour plus d'informations, consultez Création d'interfaces génériques de type variant (C#) et Variance dans les délégués (C#).

Rubriques connexes

TITRE	DESCRIPTION
Variance dans les interfaces génériques (C#)	Décrit la covariance et la contravariance dans les interfaces génériques et fournit une liste d'interfaces génériques de type Variant dans .NET.
Création d'interfaces génériques de type variant (C#)	Montre comment créer des interfaces de type variant personnalisées.
Utilisation de la variance dans les interfaces pour les collections génériques (C#)	Montre comment la prise en charge de la covariance et de la contravariance dans les interfaces IEnumerable<T> et IComparable<T> peut vous aider à réutiliser du code.
Variance dans les délégués (C#)	Décrit la covariance et la contravariance dans les délégués génériques et non génériques et fournit une liste de délégués génériques variants dans .NET.
Utilisation de la variance dans les délégués (C#)	Montre comment utiliser la prise en charge de la covariance et de la contravariance dans les délégués non génériques pour faire correspondre des signatures de méthode avec des types délégués.

TITRE	DESCRIPTION
Utilisation de la variance pour les délégués génériques Func et Action (C#)	Montre comment la prise en charge de la covariance et de la contravariance dans les délégués `Func` et `Action` peut vous aider à réutiliser du code.

Variance dans les interfaces génériques (C#)

02/11/2020 • 4 minutes to read • Edit Online

.NET Framework 4 a introduit la prise en charge de la variance pour plusieurs interfaces génériques existantes. La prise en charge de la variance permet la conversion implicite des classes qui implémentent ces interfaces.

À partir de .NET Framework 4, les interfaces suivantes sont des variantes :

- IEnumerable<T> (T est covariant)
- IEnumerator<T> (T est covariant)
- IQueryable<T> (T est covariant)
- IGrouping<TKey,TElement> (`TKey` et `TElement` sont covariants)
- IComparer<T> (T est contravariant)
- IEqualityComparer<T> (T est contravariant)
- IComparable<T> (T est contravariant)

À compter de .NET Framework 4.5, les interfaces suivantes sont des variants :

- IReadOnlyList<T> (T est covariant)
- IReadOnlyCollection<T> (T est covariant)

La covariance permet à une méthode d'avoir un type de retour plus dérivé que celui défini par le paramètre de type générique de l'interface. Pour illustrer la fonctionnalité de covariance, considérez ces interfaces génériques : `IEnumerable<Object>` et `IEnumerable<String>`. L'interface `IEnumerable<String>` n'hérite pas de l'interface `IEnumerable<Object>`. Toutefois, le type `String` hérite du type `Object` et, dans certains cas, vous pouvez assigner des objets de ces interfaces de l'un à l'autre. Ce cas de figure est présenté dans l'exemple de code suivant :

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Dans les versions antérieures de .NET Framework, ce code provoque une erreur de compilation en C# et, si `Option Strict` est activé, dans Visual Basic. Cependant, vous pouvez désormais utiliser `strings` au lieu d' `objects`, comme illustré dans l'exemple précédent, parce que l'interface IEnumerable<T> est covariante.

La contravariance permet à une méthode d'avoir des types d'argument moins dérivés que ceux spécifiés par le paramètre générique de l'interface. Pour illustrer la contravariance, supposez que vous avez créé une classe `BaseComparer` pour comparer des instances de la classe `BaseClass`. La classe `BaseComparer` implémente l'interface `IEqualityComparer<BaseClass>`. Étant donné que l'interface IEqualityComparer<T> est maintenant contravariante, vous pouvez utiliser `BaseComparer` pour comparer des instances des classes qui héritent de la classe `BaseClass`. Ce cas de figure est présenté dans l'exemple de code suivant :

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Pour obtenir d'autres exemples, consultez Utilisation de la variance dans les interfaces pour les collections génériques (C#).

La variance dans les interfaces génériques est prise en charge uniquement pour les types référence. Les types valeur ne prennent pas en charge la variance. Par exemple, `IEnumerable<int>` ne peut pas être converti implicitement en `IEnumerable<object>`, car les entiers sont représentés par un type valeur.

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Il est également important de se souvenir que les classes qui implémentent des interfaces variantes sont toujours invariantes. Par exemple, même si List<T> implémente l'interface covariante IEnumerable<T>, vous ne pouvez pas convertir implicitement `List<String>` en `List<Object>`. En voici une illustration dans l'exemple de code suivant.

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

Voir aussi

- Utilisation de la variance dans les interfaces pour les collections génériques (C#)
- Création d'interfaces génériques de type variant (C#)
- Interfaces génériques
- Variance dans les délégués (C#)

Création d'interfaces génériques de type variant (C#)

02/11/2020 • 8 minutes to read • Edit Online

Vous pouvez déclarer des paramètres de type générique dans les interfaces comme covariant ou contravariant. La *covariance* permet aux méthodes d'interface d'avoir des types de retour plus dérivés que ceux définis par les paramètres de type générique. La *contravariance* permet aux méthodes d'interface d'avoir des types d'argument moins dérivés que ceux spécifiés par les paramètres génériques. Une interface générique ayant des paramètres de type générique covariant ou contravariant est appelée *variante*.

> **NOTE**
>
> .NET Framework 4 a introduit la prise en charge de la variance pour plusieurs interfaces génériques existantes. Pour obtenir la liste des interfaces de type Variant dans .NET, consultez variance dans les interfaces génériques (C#).

Déclaration d'interfaces génériques de type variant

Vous pouvez déclarer des interfaces génériques de type variant à l'aide des mots clés `in` et `out` pour les paramètres de type générique.

> **IMPORTANT**
>
> En C#, les paramètres `ref`, `in` et `out` ne peuvent pas être de type variant. Les types valeur ne prennent pas non plus en charge la variance.

Vous pouvez déclarer un covariant de paramètre de type générique à l'aide du mot clé `out`. Le type covariant doit remplir les conditions suivantes :

- Le type est utilisé uniquement comme un type de retour de méthodes d'interface, et non pas comme un type d'arguments de méthode. L'exemple suivant en fournit une illustration dans laquelle le type `R` est déclaré covariant.

  ```
  interface ICovariant<out R>
  {
      R GetSomething();
      // The following statement generates a compiler error.
      // void SetSomething(R sampleArg);

  }
  ```

  Il existe une exception à cette règle. Si vous avez un délégué générique contravariant comme paramètre de méthode, vous pouvez utiliser le type comme paramètre de type générique pour le délégué. L'exemple suivant du type `R` illustre ce principe. Pour plus d'informations, consultez Variance dans les délégués (C#) et Utilisation de la variance pour les délégués génériques Func et Action (C#).

  ```
  interface ICovariant<out R>
  {
      void DoSomething(Action<R> callback);
  }
  ```

- Le type n'est pas utilisé comme contrainte générique pour les méthodes d'interface. C'est ce que montre le

code suivant.

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Vous pouvez déclarer un contravariant de paramètre de type générique à l'aide du mot clé `in`. Le type contravariant peut être utilisé uniquement comme type d'arguments de méthode, et non comme type de retour de méthodes d'interface. Le type contravariant peut également être utilisé pour les contraintes génériques. Le code suivant montre comment déclarer une interface de type contravariant et utiliser une contrainte générique pour l'une de ses méthodes.

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

Il est également possible de prendre en charge à la fois la covariance et la contravariance dans la même interface, mais pour des paramètres de type différent, comme indiqué dans l'exemple de code suivant.

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implémentation d'interfaces génériques de type variant

Vous implémentez des interfaces génériques de type variant dans les classes en utilisant la même syntaxe que celle utilisée pour les interfaces invariantes. L'exemple de code suivant montre comment implémenter une interface de type covariant dans une classe générique.

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Les classes qui implémentent des interfaces de type variant sont invariantes. Par exemple, prenons le code suivant.

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extension d'interfaces génériques de type variant

Quand vous étendez une interface générique de type variant, vous devez utiliser les mots clés `in` et `out` pour spécifier explicitement si l'interface dérivée prend en charge la variance. Le compilateur ne déduit pas la variance de l'interface étendue. Observons, par exemple, les interfaces suivantes.

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

Dans l'interface `IInvariant<T>`, le paramètre de type générique `T` est invariant, alors que dans `IExtCovariant<out T>`, le paramètre de type est covariant, bien que les deux interfaces étendent la même interface. La même règle est appliquée aux paramètres de type générique contravariant.

Vous pouvez créer une interface qui étend à la fois l'interface dans laquelle le paramètre de type générique `T` est covariant et l'interface dans laquelle il est contravariant si, dans l'interface étendue, le paramètre de type générique `T` est invariant. En voici une illustration dans l'exemple de code suivant.

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

Toutefois, si un paramètre de type générique `T` est déclaré covariant dans une interface, vous ne pouvez pas le déclarer contravariant dans l'interface étendue, ou vice versa. En voici une illustration dans l'exemple de code suivant.

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Éviter l'ambiguïté

Quand vous implémentez des interfaces génériques de type variant, la variance peut parfois mener à une certaine ambiguïté. Cette ambiguïté doit être évitée.

Par exemple, si vous implémentez explicitement la même interface générique de type variant avec des paramètres de type générique différents dans une classe, cela peut être source d'ambiguïté. Le compilateur ne génère pas d'erreur dans ce cas, mais il n'est pas spécifié, quelle que soit l'implémentation de l'interface, qui sera choisie au moment de l'exécution. Cette ambiguïté peut entraîner des bogues subtils dans votre code. Considérez l'exemple de code suivant.

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

Dans cet exemple, la façon dont la méthode `pets.GetEnumerator` choisit entre `Cat` et `Dog` n'est pas spécifiée. Cela peut provoquer des problèmes dans votre code.

Voir aussi

- Variance dans les interfaces génériques (C#)
- Utilisation de la variance pour les délégués génériques Func et Action (C#)

Utilisation de la variance dans les interfaces pour les collections génériques (C#)

02/11/2020 • 3 minutes to read • Edit Online

Une interface covariante permet à ses méthodes de retourner des types plus dérivés que ceux spécifiés dans l'interface. Une interface contravariante permet à ses méthodes d'accepter des paramètres de types moins dérivés que ceux spécifiés dans l'interface.

Dans .NET Framework 4, plusieurs interfaces existantes sont devenues covariantes et contravariantes. Celles-ci comprennent IEnumerable<T> et IComparable<T>. Cela vous permet de réutiliser des méthodes qui fonctionnent avec les collections génériques de types de base pour les collections de types dérivés.

Pour obtenir la liste des interfaces de type Variant dans .NET, consultez variance dans les interfaces génériques (C#).

Conversion de collections génériques

L'exemple suivant illustre les avantages de la prise en charge de la covariance dans l'interface IEnumerable<T>. La méthode `PrintFullName` accepte une collection de type `IEnumerable<Person>` comme paramètre. Toutefois, vous pouvez la réutiliser pour une collection de type `IEnumerable<Employee>`, car `Employee` hérite de `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparaison de collections génériques

L'exemple suivant illustre les avantages de la prise en charge de la contravariance dans l'interface IComparer<T>. La classe `PersonComparer` implémente l'interface `IComparer<Person>`. Toutefois, vous pouvez réutiliser cette classe pour comparer une séquence d'objets de type `Employee`, car `Employee` hérite de `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

Voir aussi

- Variance dans les interfaces génériques (C#)

Variance dans les délégués (C#)

02/11/2020 • 10 minutes to read • Edit Online

.NET Framework 3.5 a introduit la prise en charge de la variance pour faire correspondre les signatures de méthode aux types délégués pour tous les délégués dans C#. Cela signifie que vous pouvez assigner aux délégués non seulement les méthodes ayant des signatures correspondantes, mais également des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué. Cela inclut à la fois des délégués génériques et non génériques.

Par exemple, considérez le code suivant, qui a deux classes et deux délégués : génériques et non génériques.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

Quand vous créez des délégués des types `SampleDelegate` ou `SampleGenericDelegate<A, R>`, vous pouvez assigner l'une des méthodes suivantes à ces délégués.

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

L'exemple de code suivant illustre la conversion implicite entre la signature de méthode et le type délégué.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Pour obtenir d'autres exemples, consultez Utilisation de la variance dans les délégués (C#) et Utilisation de la variance pour les délégués génériques Func et Action (C#).

Variance dans les paramètres de type générique

Dans .NET Framework 4 ou version ultérieure, vous pouvez activer la conversion implicite entre les délégués afin que les délégués génériques ayant des types différents spécifiés par les paramètres de type générique puissent être assignés les uns aux autres, si les types sont hérités les uns des autres comme requis par la variance.

Pour activer la conversion implicite, vous devez déclarer explicitement les paramètres génériques dans un délégué comme covariant ou contravariant à l'aide du mot clé `in` ou `out`.

L'exemple de code suivant indique comment vous pouvez créer un délégué ayant un paramètre de type générique covariant.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Si vous utilisez uniquement la prise en charge de la variance pour faire correspondre les signatures de méthode aux types délégués et que vous n'utilisez pas les mots clés `in` et `out`, vous pouvez réaliser qu'il est parfois possible d'instancier des délégués avec des expressions ou méthodes lambda identiques, mais que vous ne pouvez pas assigner un délégué à un autre.

Dans l'exemple de code suivant, `SampleGenericDelegate<String>` ne peut pas être converti explicitement en `SampleGenericDelegate<Object>`, même si `String` hérite de `Object`. Vous pouvez résoudre ce problème en marquant le paramètre générique `T` avec le mot clé `out`.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Délégués génériques qui ont des paramètres de type Variant dans .NET

.NET Framework 4 a introduit la prise en charge de la variance pour les paramètres de type générique dans plusieurs délégués génériques existants :

- Les délégués `Action` de l'espace de noms System, par exemple, Action<T> et Action<T1,T2>
- Les délégués `Func` de l'espace de noms System, par exemple, Func<TResult> et Func<T,TResult>
- Le délégué Predicate<T>
- Le délégué Comparison<T>
- Le délégué Converter<TInput,TOutput>

Pour obtenir plus d'informations et d'exemples, consultez Utilisation de la variance pour les délégués génériques Func et Action (C#).

Déclaration de paramètres de type variant dans les délégués génériques

Si un délégué générique possède des paramètres de type générique covariant ou contravariant, il peut être désigné sous le nom de *délégué générique variant*.

Vous pouvez déclarer un paramètre de type générique covariant dans un délégué générique à l'aide du mot clé `out`. Le type covariant peut être utilisé uniquement comme type de retour de méthode et non comme type d'arguments de méthode. L'exemple de code suivant indique comment déclarer un délégué générique covariant.

```
public delegate R DCovariant<out R>();
```

Vous pouvez déclarer un paramètre de type générique contravariant dans un délégué générique à l'aide du mot clé `in`. Le type contravariant peut être utilisé uniquement comme type d'arguments de méthode et non comme type de retour de méthode. L'exemple de code suivant indique comment déclarer un délégué générique contravariant.

```
public delegate void DContravariant<in A>(A a);
```

> **IMPORTANT**
> Les paramètres `ref`, `in` et `out` en C# ne peuvent pas être marqués comme étant variants.

Il est également possible de prendre en charge à la fois la variance et la covariance dans le même délégué, mais pour des paramètres de type différents. Cela est illustré par l'exemple suivant.

```
public delegate R DVariant<in A, out R>(A a);
```

Instanciation et appel de délégués génériques variants

Vous pouvez instancier et appeler des délégués variants de la même façon que vous instanciez et appelez des délégués invariants. Dans l'exemple suivant, le délégué est instancié par une expression lambda.

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combinaison des délégués génériques variants

Ne combinez pas les délégués variants. La méthode Combine ne prend pas en charge la conversion des délégués variants et nécessite le même type pour tous les délégués. Il peut s'ensuivre une exception runtime quand vous combinez les délégués à l'aide de la méthode Combine ou de l'opérateur `+`, comme dans l'exemple de code suivant.

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variance dans les paramètres de type générique pour les types valeur et référence

La variance pour les paramètres de type générique est prise en charge uniquement pour les types référence. Par exemple, `DVariant<int>` ne peut pas être converti implicitement en `DVariant<Object>` ni `DVariant<long>` parce que l'entier est un type valeur.

L'exemple suivant montre que la variance dans les paramètres de type générique n'est pas prise en charge pour les types valeur.

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

Voir aussi

- Génériques
- Utilisation de la variance pour les délégués génériques Func et Action (C#)
- Comment combiner des délégués (délégués multicast)

Utilisation de la variance dans les délégués (C#)

02/11/2020 • 3 minutes to read • Edit Online

Quand vous assignez une méthode à un délégué, la *covariance* et la *contravariance* offrent une grande flexibilité pour la mise en correspondance d'un type délégué avec une signature de méthode. La covariance permet à une méthode d'avoir un type de retour qui est plus dérivé que celui défini dans le délégué. La contravariance autorise une méthode qui a des types de paramètres moins dérivés que ceux du type délégué.

Exemple 1 : Covariance

Description

Cet exemple montre comment vous pouvez utiliser des délégués avec des méthodes ayant des types de retour dérivés du type de retour dans la signature du délégué. Le type de données retourné par `DogsHandler` est `Dogs`, qui dérive du type `Mammals` défini dans le délégué.

Code

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Exemple 2 : Contravariance

Description

Cet exemple montre comment vous pouvez utiliser des délégués avec des méthodes ayant des paramètres dont les types sont des types de base du type de paramètre de signature de délégué. Avec la contravariance, vous pouvez maintenant utiliser un gestionnaire d'événements plutôt que des gestionnaires distincts. L'exemple suivant utilise deux délégués :

- Un délégué KeyEventHandler qui définit la signature de l'événement Button.KeyDown. Sa signature est :

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Un délégué MouseEventHandler qui définit la signature de l'événement Button.MouseClick. Sa signature est :

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

L'exemple définit un gestionnaire d'événements avec un paramètre EventArgs et s'en sert pour gérer les événements `Button.KeyDown` et `Button.MouseClick`. C'est possible parce que EventArgs est un type de base de KeyEventArgs et MouseEventArgs.

Code

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

Voir aussi

- Variance dans les délégués (C#)
- Utilisation de la variance pour les délégués génériques Func et Action (C#)

Utilisation de la variance pour les délégués génériques Func et Action (C#)

02/11/2020 • 2 minutes to read • Edit Online

Ces exemples montrent comment utiliser la covariance et la contravariance dans les délégués génériques `Func` et `Action` pour permettre de réutiliser des méthodes et fournir davantage de flexibilité dans votre code.

Pour plus d'informations sur la covariance et la contravariance, consultez Variance dans les délégués (C#).

Utilisation des délégués avec les paramètres de type covariant

L'exemple suivant illustre les avantages de la prise en charge de la covariance dans les délégués génériques `Func`. La méthode `FindByTitle` prend un paramètre de type `String` et retourne un objet de type `Employee`. Toutefois, vous pouvez assigner cette méthode au délégué `Func<String, Person>`, car `Employee` hérite de `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Utilisation des délégués avec les paramètres de type contravariant

L'exemple suivant illustre les avantages de la prise en charge de la contravariance dans les délégués génériques `Action`. La méthode `AddToContacts` prend un paramètre de type `Person`. Toutefois, vous pouvez assigner cette méthode au délégué `Action<Employee>`, car `Employee` hérite de `Person`.

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Voir aussi

- Covariance et contravariance (C#)
- Génériques

Arborescences d'expressions (C#)

02/11/2020 • 7 minutes to read • Edit Online

Les arborescences d'expressions représentent du code dans une structure de données arborescente, où chaque nœud est une expression, par exemple un appel de méthode ou une opération binaire comme `x < y`.

Vous pouvez compiler et exécuter du code représenté par des arborescences d'expressions. Ceci permet la modification dynamique du code exécutable, l'exécution de requêtes LINQ dans différentes bases de données et la création de requêtes dynamiques. Pour plus d'informations sur les arborescences d'expressions dans LINQ, consultez comment utiliser des arborescences d'expression pour générer des requêtes dynamiques (C#).

Les arborescences d'expressions sont également utilisées dans le Dynamic Language Runtime (DLR) pour assurer l'interopérabilité entre les langages dynamiques et .NET, ainsi que pour permettre aux Writers de compilateur d'émettre des arborescences d'expressions au lieu du langage MSIL (Microsoft Intermediate Language). Pour plus d'informations sur le DLR, consultez Vue d'ensemble du Dynamic Language Runtime.

Le compilateur C# ou Visual Basic peut créer pour vous une arborescence d'expressions basée sur une expression lambda anonyme. Vous pouvez aussi créer manuellement des arborescences d'expressions en utilisant l'espace de noms System.Linq.Expressions.

Création d'arborescences d'expressions à partir d'expressions lambda

Quand une expression lambda est affectée à une variable de type Expression<TDelegate>, le compilateur produit du code pour générer une arborescence d'expressions qui représente l'expression lambda.

Le compilateur C# peut générer des arborescences d'expressions seulement à partir d'expressions lambda (ou de lambdas sur une seule ligne). Ils ne peuvent pas analyser des lambdas d'instruction (ou lambdas multilignes). Pour plus d'informations sur les expressions lambda en C#, consultez Expressions lambda.

Les exemples de code suivants montrent comment le compilateur C# crée une arborescence d'expressions qui représente l'expression lambda `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Création d'arborescences d'expressions à l'aide de l'API

Pour créer des arborescences d'expressions à l'aide de l'API, utilisez la classe Expression. Cette classe contient des méthodes de fabrique statiques qui créent des nœuds d'arborescence d'expressions de types spécifiques, par exemple ParameterExpression, qui représente une variable ou un paramètre, ou MethodCallExpression, qui représente un appel de méthode. ParameterExpression, MethodCallExpression et les autres types spécifiques à une expression sont également définis dans l'espace de noms System.Linq.Expressions. Ces types dérivent du type abstrait Expression.

L'exemple de code suivant montre comment créer une arborescence d'expressions qui représente l'expression lambda `num => num < 5` à l'aide de l'API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

Dans le .NET Framework 4 ou ultérieur, l'API des arborescences d'expressions prend également en charge les affectations et les expressions de flux de contrôle, comme les boucles, les blocs conditionnels et les blocs `try-catch`. À l'aide de l'API, vous pouvez créer des arborescences d'expressions qui sont plus complexes que celles qui peuvent être créées à partir d'expressions lambda par le compilateur C#. L'exemple suivant montre comment créer une arborescence d'expressions qui calcule la factorielle d'un nombre.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
        Expression.Loop(
    // Adding a conditional block into the loop.
           Expression.IfThenElse(
    // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
    // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
    // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
    // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

Pour plus d'informations, consultez Génération de méthodes dynamiques avec des arborescences d'expressions dans Visual Studio 2010 (ou version ultérieure).

Analyse des arborescences d'expressions

L'exemple de code suivant montre comment l'arborescence d'expressions qui représente l'expression lambda `num => num < 5` peut être décomposée selon ses différentes parties.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Immuabilité des arborescences d'expressions

Les arborescences d'expressions doivent être immuables. Cela signifie que si vous voulez modifier une arborescence d'expressions, vous devez construire une nouvelle arborescence d'expressions en copiant l'arborescence existante et en y remplaçant les nœuds. Vous pouvez utiliser un visiteur d'arborescence d'expressions pour parcourir l'arborescence d'expressions existante. Pour plus d'informations, consultez Comment modifier les arborescences d'expressions (C#).

Compilation des arborescences d'expressions

Le type Expression<TDelegate> fournit la méthode Compile qui compile le code représenté par une arborescence d'expressions en un délégué exécutable.

L'exemple de code suivant montre comment compiler une arborescence d'expressions et exécuter le code résultant.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

Pour plus d'informations, consultez comment exécuter des arborescences d'expressions (C#).

Voir aussi

- System.Linq.Expressions
- Comment exécuter des arborescences d'expressions (C#)
- Comment modifier les arborescences d'expressions (C#)
- Expressions lambda
- Vue d'ensemble du Dynamic Language Runtime
- Concepts de programmation (C#)

Comment exécuter des arborescences d'expressions (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette rubrique montre comment exécuter une arborescence d'expressions. L'exécution d'une arborescence d'expressions peut retourner une valeur, ou elle peut simplement effectuer une action telle que l'appel d'une méthode.

Seules les arborescences d'expressions qui représentent des expressions lambda peuvent être exécutées. Les arborescences d'expressions qui représentent des expressions lambda peuvent être de type LambdaExpression ou Expression<TDelegate>. Pour exécuter ces arborescences d'expressions, appelez la méthode Compile pour créer un délégué exécutable, puis appelez le délégué.

> **NOTE**
> Si le type du délégué n'est pas connu, autrement dit si l'expression lambda est de type LambdaExpression et non Expression<TDelegate>, vous devez appeler la méthode DynamicInvoke sur le délégué au lieu de l'appeler directement.

Si une arborescence d'expressions ne représente pas une expression lambda, vous pouvez créer une expression lambda ayant l'arborescence d'expressions d'origine comme corps, en appelant la méthode Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>). Ensuite, vous pouvez exécuter l'expression lambda comme décrit plus haut dans cette section.

Exemple

L'exemple de code suivant montre comment exécuter une arborescence d'expressions qui représente l'élévation d'un nombre à une puissance en créant une expression lambda et en l'exécutant. Le résultat, qui représente le nombre élevé à la puissance, est affiché.

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Compilation du code

- Incluez l'espace de noms System.Linq.Expressions.

Voir aussi

- Arborescences d'expressions (C#)
- Comment modifier les arborescences d'expressions (C#)

Comment modifier les arborescences d'expressions (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cette rubrique montre comment modifier une arborescence d'expressions. Les arborescences d'expressions sont immuables, ce qui signifie qu'elles ne peuvent pas être modifiées directement. Pour changer une arborescence d'expressions, vous devez créer une copie d'une arborescence d'expressions existante et, quand vous créez la copie, apporter les modifications nécessaires. Vous pouvez utiliser la classe ExpressionVisitor pour parcourir une arborescence d'expressions existante et copier chaque nœud visité.

Pour modifier une arborescence d'expressions

1. Créez un projet d' **application console** .
2. Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`.
3. Ajoutez la classe `AndAlsoModifier` à votre projet.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

   Cette classe hérite de la classe `AND` et est spécialisée pour modifier les expressions qui représentent des opérations ExpressionVisitor conditionnelles. Elle convertit ces opérations d'un `AND` conditionnel en un `OR` conditionnel. Pour ce faire, la classe substitue la méthode VisitBinary du type de base, car les expressions `AND` conditionnelles sont représentées sous forme d'expressions binaires. Dans la méthode `VisitBinary`, si l'expression passée représente une opération `AND` conditionnelle, le code construit une nouvelle expression qui contient l'opérateur `OR` conditionnel au lieu de l'opérateur `AND` conditionnel. Si l'expression passée à `VisitBinary` ne représente pas une opération `AND` conditionnelle, la méthode emploie l'implémentation de classe de base. Les méthodes de classe de base construisent des nœuds qui sont comme les arborescences d'expressions passées, mais les nœuds voient leurs sous-arborescences remplacées par les arborescences d'expressions produites de manière récursive par le visiteur.

4. Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`.
5. Ajoutez le code à la méthode `Main` dans le fichier Program.cs pour créer une arborescence d'expressions, et

passez-le à la méthode qui le modifiera.

```
Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

Le code crée une expression qui contient une opération `AND` conditionnelle. Il crée ensuite une instance de la classe `AndAlsoModifier` et passe l'expression à la méthode `Modify` de cette classe. Les arborescences d'expressions d'origine et modifiée sont toutes deux générées pour montrer la modification.

6. Compilez et exécutez l'application.

Voir aussi

- Comment exécuter des arborescences d'expressions (C#)
- Arborescences d'expressions (C#)

Comment utiliser des arborescences d'expression pour générer des requêtes dynamiques (C#)

02/11/2020 • 5 minutes to read • Edit Online

Dans LINQ, des arborescences d'expressions sont utilisées pour représenter des requêtes structurées qui ciblent des sources de données qui implémentent IQueryable<T>. Par exemple, le fournisseur LINQ implémente l'interface IQueryable<T> pour interroger des magasins de données relationnelles. Le compilateur C# compile les requêtes qui ciblent de telles sources de données dans du code qui génère une arborescence d'expressions lors de l'exécution. Le fournisseur de requêtes peut ensuite parcourir la structure de données de l'arborescence d'expressions et la traduire en un langage de requête qui convienne à la source de données.

Des arborescences d'expressions sont aussi utilisées dans LINQ pour représenter des expressions lambda assignées à des variables de type Expression<TDelegate>.

Cette rubrique explique comment utiliser des arborescences d'expressions pour créer des requêtes LINQ dynamiques. Les requêtes dynamiques sont utiles lorsque les caractéristiques d'une requête ne sont pas connues au moment de la compilation. Par exemple, une application peut fournir une interface utilisateur qui permet aux utilisateurs de spécifier un ou plusieurs prédicats pour filtrer des données. Si vous souhaitez utiliser LINQ pour interroger des données, ce type d'application doit utiliser des arborescences d'expressions pour créer la requête LINQ au moment de l'exécution.

Exemple

L'exemple suivant montre comment utiliser des arborescences d'expressions pour construire une requête et l'exécuter sur une source de données `IQueryable`. Le code génère une arborescence d'expressions pour représenter la requête suivante :

```
companies.Where(company => (company.ToLower() == "coho winery" || company.Length > 16))
         .OrderBy(company => company)
```

Les méthodes de fabrique de l'espace de noms System.Linq.Expressions sont utilisées pour créer des arborescences d'expressions qui représentent les expressions qui composent l'ensemble de la requête. Les expressions qui représentent des appels aux méthodes d'opérateur de requête standard font référence aux implémentations Queryable de ces méthodes. L'arborescence d'expression finale est passée à l'implémentation CreateQuery<TElement>(Expression) du fournisseur de la source de données `IQueryable` pour créer une requête exécutable de type `IQueryable`. Les résultats sont obtenus par l'énumération de cette variable de requête.

```
// Add a using directive for System.Linq.Expressions.

string[] companies = { "Consolidated Messenger", "Alpine Ski House", "Southridge Video", "City Power & Light",
                    "Coho Winery", "Wide World Importers", "Graphic Design Institute", "Adventure Works",
                    "Humongous Insurance", "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
                    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
                    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee" };

// The IQueryable data to query.
IQueryable<String> queryableData = companies.AsQueryable<string>();

// Compose the expression tree that represents the parameter to the predicate.
ParameterExpression pe = Expression.Parameter(typeof(string), "company");

// ***** Where(company => (company.ToLower() == "coho winery" || company.Length > 16)) *****
```

```
// Create an expression tree that represents the expression 'company.ToLower() == "coho winery"'.
Expression left = Expression.Call(pe, typeof(string).GetMethod("ToLower", System.Type.EmptyTypes));
Expression right = Expression.Constant("coho winery");
Expression e1 = Expression.Equal(left, right);

// Create an expression tree that represents the expression 'company.Length > 16'.
left = Expression.Property(pe, typeof(string).GetProperty("Length"));
right = Expression.Constant(16, typeof(int));
Expression e2 = Expression.GreaterThan(left, right);

// Combine the expression trees to create an expression tree that represents the
// expression '(company.ToLower() == "coho winery" || company.Length > 16)'.
Expression predicateBody = Expression.OrElse(e1, e2);

// Create an expression tree that represents the expression
// 'queryableData.Where(company => (company.ToLower() == "coho winery" || company.Length > 16))'
MethodCallExpression whereCallExpression = Expression.Call(
    typeof(Queryable),
    "Where",
    new Type[] { queryableData.ElementType },
    queryableData.Expression,
    Expression.Lambda<Func<string, bool>>(predicateBody, new ParameterExpression[] { pe }));
// ***** End Where *****

// ***** OrderBy(company => company) *****
// Create an expression tree that represents the expression
// 'whereCallExpression.OrderBy(company => company)'
MethodCallExpression orderByCallExpression = Expression.Call(
    typeof(Queryable),
    "OrderBy",
    new Type[] { queryableData.ElementType, queryableData.ElementType },
    whereCallExpression,
    Expression.Lambda<Func<string, string>>(pe, new ParameterExpression[] { pe }));
// ***** End OrderBy *****

// Create an executable query from the expression tree.
IQueryable<string> results = queryableData.Provider.CreateQuery<string>(orderByCallExpression);

// Enumerate the results.
foreach (string company in results)
    Console.WriteLine(company);

/*  This code produces the following output:

    Blue Yonder Airlines
    City Power & Light
    Coho Winery
    Consolidated Messenger
    Graphic Design Institute
    Humongous Insurance
    Lucerne Publishing
    Northwind Traders
    The Phone Company
    Wide World Importers
*/
```

Ce code utilise un nombre fixe d'expressions dans le prédicat qui est passé à la méthode `Queryable.Where`. Toutefois, vous pouvez écrire une application qui combine un nombre variable d'expressions de prédicat dépendant des entrées utilisateur. Vous pouvez également varier les opérateurs de requête standard qui sont appelés dans la requête, en fonction des entrées utilisateur.

Compilation du code

- Incluez l'espace de noms System.Linq.Expressions.

Voir aussi

- Arborescences d'expressions (C#)
- Comment exécuter des arborescences d'expressions (C#)
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution

Débogage d'arborescences d'expression dans Visual Studio (C#)

02/11/2020 • 2 minutes to read • Edit Online

Vous pouvez analyser la structure et le contenu d'arborescences d'expression quand vous déboguez vos applications. Pour obtenir un rapide aperçu de l'arborescence d'expression, vous pouvez utiliser la propriété `DebugView`, qui représente des arborescences d'expression en utilisant une syntaxe spéciale. (Notez que `DebugView` est disponible uniquement en mode débogage.)

Étant donné que `DebugView` est une chaîne, vous pouvez utiliser le visualiseur de texte intégré pour l'afficher sur plusieurs lignes, en sélectionnant **Visualiseur de texte** à partir de l'icône de loupe située à côté de l'étiquette `DebugView`.

Vous pouvez également installer et utiliser un visualiseur personnalisé pour les arborescences d'expression, notamment :

- Les expressions lisibles (licence MIT, disponible au Visual Studio Marketplace), affichent l'arborescence de l'expression en tant que code C# thématique, avec diverses options de rendu :

- Le visualiseur de l' arborescence d'expression (licence MIT) fournit une arborescence de l'arborescence de l'expression et de ses nœuds individuels :

Pour ouvrir un visualiseur pour une arborescence d'expressions

1. Cliquez sur l'icône de loupe qui s'affiche en regard de l'arborescence d'expression dans **DataTips**, dans une fenêtre **Espion**, dans la fenêtre **Automatique** ou dans la fenêtre **Variables locales**.

   Une liste de visualiseurs disponibles s'affiche :

2. Cliquez sur le visualiseur à utiliser.

Voir aussi

- Arborescences d'expressions (C#)
- Débogage dans Visual Studio
- Créer des visualiseurs personnalisés
- `DebugView` stockéesyntaxe

Syntaxe de `DebugView`

18/03/2020 • 3 minutes to read • Edit Online

La propriété `DebugView` (disponible uniquement lors du débogage) fournit un rendu de la chaîne d'arborescences d'expression. La plupart de la syntaxe est assez simple à comprendre ; les cas spéciaux sont décrits dans les sections suivantes.

Chaque exemple est suivi d'un commentaire en bloc contenant le `DebugView`.

ParameterExpression

Les noms de variables System.Linq.Expressions.ParameterExpression s'affichent avec le symbole `$` au début.

Si un paramètre n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `$var1` ou `$var2`.

Exemples

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Pour les objets System.Linq.Expressions.ConstantExpression qui représentent des valeurs entières, des chaînes et des valeurs `null`, la valeur de la constante est affichée.

Pour les types numériques comportant des suffixes standard comme littéraux C#, le suffixe est ajouté à la valeur. Le tableau suivant indique les suffixes associés aux différents types numériques.

TYPE	MOT CLÉ	SUFFIXE
System.UInt32	uint	U
System.Int64	Long	L
System.UInt64	Ulong	UL
System.Double	Double	D
System.Single	Flotteur	F
System.Decimal	Decimales	M

Exemples

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Si le type d'un objet System.Linq.Expressions.BlockExpression diffère du type de la dernière expression du bloc, le type est affiché entre crochets pointus (`<` et `>`). Sinon, le type de l'objet BlockExpression n'est pas affiché.

Exemples

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

Les objets System.Linq.Expressions.LambdaExpression sont affichés avec leurs types délégués.

Si une expression lambda n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `#Lambda1` ou `#Lambda2`.

Exemples

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Si vous spécifiez une valeur par défaut pour l'objet System.Linq.Expressions.LabelExpression, cette valeur est

affichée avant l'objet System.Linq.Expressions.LabelTarget.

Le jeton `.Label` indique le début de l'étiquette. Le jeton `.LabelTarget` indique la destination de la cible à laquelle accéder.

Si une étiquette n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `#Label1` ou `#Label2`.

Exemples

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Opérateurs Checked

Des opérateurs activés sont affichés avec le symbole `#` en regard de l'opérateur. Par exemple, l'opérateur d'addition checked est affiché sous la forme `#+`.

Exemples

```
Expression expr = Expression.AddChecked( Expression.Constant(1), Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0), typeof(int));
/*
    #(System.Int32)10D
*/
```

Itérateurs (C#)

02/11/2020 • 12 minutes to read • Edit Online

Un *itérateur* peut être utilisé pour parcourir des collections, comme des listes et des tableaux.

Une méthode d'itérateur ou un accesseur `get` effectue une itération personnalisée sur une collection. Une méthode d'itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution est redémarrée à partir de cet emplacement lors de l'appel suivant de la fonction d'itérateur.

Vous consommez un itérateur à partir du code client en utilisant une instruction foreach ou une requête LINQ.

Dans l'exemple suivant, la première itération de la boucle `foreach` fait que l'exécution continue dans la méthode d'itérateur `SomeNumbers`, jusqu'à ce que la première instruction `yield return` soit atteinte. Cette itération retourne la valeur 3, et l'emplacement actif dans la méthode d'itérateur est mémorisé. Dans l'itération suivante de la boucle, l'exécution de la méthode d'itérateur continue là où elle s'était arrêtée, et s'arrête de nouveau quand elle atteint une instruction `yield return`. Cette itération retourne la valeur 5, et l'emplacement actif dans la méthode d'itérateur est mémorisé. La boucle se termine quand la fin de la méthode d'itérateur est atteinte.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

Le type de retour d'une méthode d'itérateur ou de l'accesseur `get` peut être IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Utilisez une instruction `yield break` pour terminer l'itération.

> **NOTE**
>
> Tous les exemples de cette rubrique, à l'exception de l'exemple de l'itérateur simple, incluent des directives using pour les espaces de noms `System.Collections` et `System.Collections.Generic`.

Itérateur simple

L'exemple suivant comprend une seule instruction `yield return` qui se trouve dans une boucle for. Dans `Main`, chaque itération du corps d'instruction `foreach` crée un appel à la fonction d'itérateur, qui poursuit avec l'instruction `yield return` suivante.

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Création d'une classe de collection

Dans l'exemple suivant, la classe `DaysOfTheWeek` implémente l'interface IEnumerable, ce qui nécessite la méthode GetEnumerator. Le compilateur appelle implicitement la méthode `GetEnumerator`, qui retourne un IEnumerator.

La méthode `GetEnumerator` retourne chaque chaîne une à la fois en utilisant l'instruction `yield return`.

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

L'exemple suivant crée une classe `Zoo` qui contient une collection d'animaux.

L'instruction `foreach` qui fait référence à l'instance de classe (`theZoo`) appelle implicitement la méthode `GetEnumerator`. Les instructions `foreach` qui font référence aux propriétés `Birds` et `Mammals` utilisent la méthode d'itérateur nommée `AnimalsForType`.

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods
```

```
    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Utilisation d'itérateurs avec une liste générique

Dans l'exemple suivant, la classe générique Stack<T> implémente l'interface générique IEnumerable<T>. La méthode Push affecte des valeurs à un tableau de type `T`. La méthode GetEnumerator retourne les valeurs de tableau à l'aide de l'instruction `yield return`.

Outre la méthode GetEnumerator générique, la méthode GetEnumerator non générique doit également être implémentée. C'est parce que IEnumerable<T> hérite de IEnumerable. L'implémentation non générique s'en remet à l'implémentation générique.

L'exemple utilise des itérateurs nommés pour prendre en charge différentes façons d'itérer au sein de la même collection de données. Ces itérateurs nommés sont les propriétés `TopToBottom` et `BottomToTop`, et la méthode `TopN`.

La propriété `BottomToTop` utilise un itérateur dans un accesseur `get`.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0
```

```
    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;
```

```
        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Informations sur la syntaxe

Un itérateur peut être une méthode ou un accesseur `get`. Un itérateur ne peut pas être présent dans un événement, un constructeur d'instance, un constructeur statique ou un finaliseur statique.

Une conversion implicite doit exister du type d'expression dans l' `yield return` instruction à l'argument de type pour le `IEnumerable<T>` retourné par l'itérateur.

En C#, une méthode d'itérateur ne peut avoir aucun paramètre `in`, `ref` ou `out`.

En C#, `yield` n'est pas un mot réservé et a une signification spéciale uniquement lorsqu'il est utilisé avant un `return` `break` mot clé ou.

Implémentation technique

Bien que vous écriviez un itérateur comme une méthode, le compilateur le traduit en une classe imbriquée, qui est en réalité une machine à états. Cette classe fait le suivi de la position de l'itérateur tant que la boucle `foreach` du code client se poursuit.

Pour voir ce que fait le compilateur, vous pouvez utiliser l'outil Ildasm.exe et afficher le code du langage intermédiaire Microsoft généré pour une méthode d'itérateur.

Quand vous créez un itérateur pour une classe ou un struct, vous n'avez pas besoin d'implémenter l'ensemble de l'interface IEnumerator. Quand le compilateur détecte l'itérateur, il génère automatiquement les méthodes `Current`, `MoveNext` et `Dispose` de l'interface IEnumerator ou IEnumerator<T>.

À chaque itération successive de la boucle `foreach` (ou de l'appel direct à `IEnumerator.MoveNext`), le corps du code de l'itérateur suivant reprend après l'instruction `yield return` précédente. Il continue ensuite jusqu'à l'instruction `yield return` suivante, jusqu'à atteindre la fin du corps de l'itérateur ou jusqu'à rencontrer une instruction `yield break`.

Les itérateurs ne prennent pas en charge la méthode IEnumerator.Reset. Pour réitérer à partir du début, vous devez obtenir un nouvel itérateur. L'appel de Reset sur l'itérateur retourné par une méthode d'itérateur lève un NotSupportedException.

Pour plus d'informations, consultez la spécification du langage C#.

Utilisation d'itérateurs

Les itérateurs vous permettent de conserver la simplicité d'une boucle `foreach` quand vous avez besoin d'utiliser un code complexe pour remplir la séquence d'une liste. Ceci peut être utile quand vous voulez faire les opérations suivantes :

- Modifier la séquence de la liste après la première itération de la boucle `foreach`.
- Éviter de charger entièrement une grande liste avant la première itération d'une boucle `foreach`. Une extraction paginée pour charger un lot de lignes d'une table est un exemple. Un autre exemple est la EnumerateFiles méthode, qui implémente les itérateurs dans .net.

- Encapsuler la création de la liste dans l'itérateur. Dans la méthode d'itérateur, vous pouvez créer la liste et générer ensuite chaque résultat dans une boucle.

Voir aussi

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- yield
- Utiliser foreach avec des tableaux
- Génériques

LINQ (Language-Integrated Query)

02/11/2020 • 7 minutes to read • Edit Online

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements. Vous pouvez écrire des requêtes pour des collections d'objets fortement typées à l'aide de mots clés de langage et d'opérateurs familiers. La famille de technologies LINQ fournit une expérience de requête cohérente pour les objets (LINQ to Objects), pour les bases de données relationnelles (LINQ to SQL) et pour le code XML (LINQ to XML).

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans des bases de données SQL, des jeux de données ADO.NET, des documents et des flux XML et des collections .NET.

Vous pouvez écrire des requêtes LINQ en C# pour des bases de données SQL Server, des documents XML, des jeux de données ADO.NET et toute collection d'objets prenant en charge IEnumerable ou l'interface générique IEnumerable<T>. La prise en charge LINQ est également fournie par des tierces parties pour de nombreux services web et autres implémentations de base de données.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

L'illustration suivante de Visual Studio montre une requête LINQ partiellement terminée, exécutée sur une

base de données SQL Server, en C# et en Visual Basic, avec un contrôle de type complet et une prise en charge IntelliSense :

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.
- Les expressions de requête sont faciles à maîtriser, car elles utilisent de nombreuses constructions familières du langage C#.
- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.
- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.
- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.
- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.
- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez syntaxe de requête et syntaxe de méthode dans LINQ.
- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à

la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Introduction aux requêtes LINQ (C#)

02/11/2020 • 11 minutes to read • Edit Online

Une *requête* est une expression qui récupère des données à partir d'une source de données. Les requêtes sont généralement exprimées dans un langage de requête spécialisé. Les différents langages ont été développés au fil du temps pour les différents types de sources de données, par exemple, SQL pour les bases de données relationnelles et XQuery pour XML. Les développeurs devaient donc apprendre un nouveau langage de requête pour chaque nouveau type de source de données ou format de données qu'ils devaient prendre en charge. LINQ simplifie cette situation en proposant un modèle cohérent pour travailler avec des données dans différents types de sources et de formats de données. Dans une requête LINQ, vous travaillez toujours avec des objets. Vous utilisez les mêmes modèles de codage de base pour interroger et transformer des données dans des documents XML, des bases de données SQL, des jeux de données ADO.NET, des collections .NET et tout autre format pour lequel un fournisseur LINQ est disponible.

Les trois parties d'une opération de requête

Toutes les opérations de requête LINQ se composent de trois actions distinctes :

1. Obtention de la source de données
2. Création de la requête
3. exécutez la requête.

L'exemple suivant montre comment les trois parties d'une opération de requête sont exprimées dans le code source. Cet exemple utilise un tableau d'entiers comme source de données pour des raisons pratiques. Toutefois, ces mêmes concepts s'appliquent également aux autres sources de données. Le reste de la rubrique fait référence à cet exemple.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

L'illustration suivante montre l'intégralité de l'opération de requête. Dans LINQ, l'exécution de la requête est distincte de la requête elle-même. En d'autres termes, vous n'avez récupéré aucune donnée en créant simplement une variable de requête.

Source de données

Dans l'exemple précédent, comme la source de données est un tableau, elle prend en charge implicitement l'interface générique IEnumerable<T>. Cela signifie qu'elle peut être interrogée avec LINQ. Une requête est exécutée dans une instruction `foreach`, et `foreach` nécessite IEnumerable ou IEnumerable<T>. Les types qui prennent en charge IEnumerable<T> ou une interface dérivée, comme l'interface générique IQueryable<T>, sont appelés des *types requêtables*.

Un type interrogeable ne nécessite aucune modification ni traitement spécial pour servir de source de données LINQ. Si les données sources ne sont pas déjà en mémoire en tant que type interrogeable, le fournisseur LINQ doit le représenter comme tel. Par exemple, LINQ to XML charge un document XML dans un type XElement requêtable :

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Avec LINQ to SQL , vous créez tout d'abord un mappage objet/relationnel au moment du design, soit manuellement, soit à l'aide des outils de LINQ to SQL dans Visual Studio. Vous écrivez vos requêtes pour des objets. Ensuite, au moment de l'exécution, LINQ to SQL gère la communication avec la base de données. Dans l'exemple suivant, `Customers` représente une table spécifique de la base de données et le type du résultat de la requête, IQueryable<T>, dérive de IEnumerable<T>.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Pour plus d'informations sur la création de types spécifiques de sources de données, consultez la documentation relative aux différents fournisseurs LINQ. Toutefois, la règle de base est très simple : une source de données LINQ est un objet qui prend en charge l' IEnumerable<T> interface générique ou une interface qui hérite de celui-ci.

> **NOTE**
> Les types tels que ArrayList qui prennent en charge l' IEnumerable interface non générique peuvent également être utilisés comme source de données Linq. Pour plus d'informations, consultez Comment interroger un ArrayList avec LINQ (C#).

La requête

La requête spécifie les informations à récupérer à partir de la ou des sources de données. Si vous le souhaitez, une requête peut également spécifier la manière dont ces informations doivent être triées, regroupées et mises en forme avant d'être retournées. Une requête est stockée dans une variable de requête et initialisée avec une expression de requête. Pour faciliter l'écriture de requêtes, le langage C# propose désormais une nouvelle syntaxe de requête.

La requête de l'exemple précédent retourne tous les nombres pairs du tableau d'entiers. L'expression de requête contient trois clauses : `from`, `where` et `select`. (Si vous êtes familiarisé avec SQL, vous aurez remarqué que l'ordre des clauses est inversé par rapport à l'ordre dans SQL.) La `from` clause spécifie la source de données, la `where` clause applique le filtre et la `select` clause spécifie le type des éléments retournés. Celles-ci et les autres clauses de requête sont abordées en détail dans la section Language Integrated Query (LINQ) . Pour le moment, le point important est que dans LINQ, la variable de requête elle-même n'effectue aucune action et ne retourne aucune donnée. Elle stocke simplement les informations qui seront nécessaires pour produire des résultats lors de l'exécution ultérieure de la requête. Pour plus d'informations sur la construction des requêtes en arrière-plan, consultez Vue d'ensemble des opérateurs de requête standard (C#).

> **NOTE**
> Les requêtes peuvent également être exprimées à l'aide d'une syntaxe de méthode. Pour plus d'informations, consultez Syntaxe de requête et syntaxe de méthode dans LINQ.

Exécution des requêtes

Exécution différée

Comme indiqué précédemment, la variable de requête stocke uniquement les commandes de requête. L'exécution de la requête est différée jusqu'à ce que vous itériez au sein de la variable de requête dans une instruction `foreach`. Ce concept est appelé *exécution différée* et est illustré dans l'exemple suivant :

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

C'est dans l'instruction `foreach` que les résultats de requête sont récupérés. Par exemple, dans la requête précédente, la variable d'itération `num` contient chaque valeur (une à la fois) de la séquence retournée.

Étant donné que la variable de requête ne contient jamais les résultats de requête, vous pouvez l'exécuter aussi souvent que vous le voulez. Par exemple, vous disposez peut-être d'une base de données qui est constamment mise à jour par une application distincte. Vous pouvez créer dans cette dernière une requête dans le but d'extraire les données les plus récentes, puis l'exécuter à intervalle régulier. Vous obtiendrez à chaque fois des résultats différents.

Forcer l'exécution immédiate

Les requêtes qui exécutent des fonctions d'agrégation sur une plage d'éléments sources doivent d'abord itérer

au sein de ces éléments. Ces requêtes sont par exemple `Count`, `Max`, `Average` et `First`. Elles s'exécutent sans instruction `foreach` explicite, car les requêtes doivent utiliser `foreach` pour retourner un résultat. Notez également que ces types de requêtes retournent une seule valeur, et non une collection `IEnumerable`. La requête suivante retourne un nombre de chiffres pairs du tableau source :

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Pour forcer l'exécution immédiate de n'importe quelle requête et mettre en cache ses résultats, vous pouvez appeler les méthodes ToList ou ToArray.

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Vous pouvez également forcer l'exécution en plaçant la boucle `foreach` immédiatement après l'expression de requête. Toutefois, en appelant `ToList` ou `ToArray`, vous mettez également en cache toutes les données dans un même objet de collection.

Voir aussi

- Mise en route de LINQ en C#
- Procédure pas à pas : écriture de requêtes en C#
- LINQ (Language-Integrated Query)
- foreach, in
- Mots clés de requête (LINQ)

LINQ et types génériques (C#)

02/11/2020 • 4 minutes to read • Edit Online

Les requêtes LINQ sont basées sur des types génériques, qui ont été introduits dans la version 2,0 de .NET Framework. Vous ne devez pas avoir une connaissance approfondie des génériques avant de commencer à écrire des requêtes. Il peut cependant être important de comprendre deux concepts de base :

1. Quand vous créez une instance d'une classe de collection générique comme List<T>, vous remplacez le « T » par le type des objets contenus dans la liste. Par exemple, une liste de chaînes est exprimée sous la forme `List<string>`, et une liste d'objets `Customer` est exprimée sous la forme `List<Customer>`. Une liste générique est fortement typée et offre de nombreux avantages par rapport aux collections qui stockent leurs éléments en tant que Object. Si vous essayez d'ajouter un `Customer` à un `List<string>`, vous obtenez une erreur à la compilation. Il est facile d'utiliser des collections génériques, car vous ne devez pas effectuer un cast de type à l'exécution.

2. IEnumerable<T> est l'interface qui permet l'énumération des classes de collection génériques avec l'instruction `foreach`. Les classes de collection génériques prennent en charge IEnumerable<T> de la même façon que les classes de collection non génériques telles que ArrayList prennent en charge IEnumerable.

Pour plus d'informations sur les génériques, consultez Génériques.

Variables IEnumerable<T> dans les requêtes LINQ

Les variables de requête LINQ sont typées comme IEnumerable<T> ou un type dérivé tel que IQueryable<T> . Quand vous voyez une variable de requête typée en `IEnumerable<Customer>`, cela signifie simplement que la requête génère à l'exécution une séquence de zéro ou plusieurs objets `Customer`.

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Pour plus d'informations, consultez relations de types dans les opérations de requête LINQ.

Laisser le compilateur gérer les déclarations de types génériques

Si vous préférez, vous pouvez éviter une syntaxe générique en utilisant le mot clé var. Le mot clé `var` indique au compilateur d'inférer le type d'une variable de requête en examinant la source de données spécifiée dans la clause `from`. L'exemple suivant génère le même code compilé que l'exemple précédent :

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Le mot clé `var` est utile quand le type de la variable est évident ou quand il est particulièrement important de spécifier explicitement des types génériques imbriqués, comme ceux qui sont produits par des requêtes de groupe. D'une façon générale, si vous utilisez `var`, sachez qu'il peut rendre votre code plus difficile à lire par d'autres développeurs. Pour plus d'informations, consultez Variables locales implicitement typées.

Voir aussi

- Génériques

Opérations de requête LINQ de base (C#)

02/11/2020 • 9 minutes to read • Edit Online

Cette rubrique fournit une brève introduction aux expressions de requête LINQ et à certains genres classiques d'opérations que vous effectuez dans une requête. Vous trouverez des informations plus détaillées dans les rubriques suivantes :

Expressions de requête LINQ

Présentation des opérateurs de requête standard (C#)

Procédure pas à pas : écriture de requêtes en C#

> **NOTE**
> Si vous êtes déjà familiarisé avec un langage de requête tel que SQL ou XQuery, vous pouvez ignorer la plupart des informations de cette rubrique. `from` Pour en savoir plus sur l'ordre des clauses dans les expressions de requête LINQ, consultez la section suivante.

Obtention d'une source de données

Dans une requête LINQ, la première étape consiste à spécifier la source de données. Dans C#, comme dans la plupart des langages de programmation, une variable doit être déclarée avant de pouvoir être utilisée. Dans une requête LINQ, la `from` clause est d'abord introduite pour introduire la source de données (`customers`) et la *variable de portée* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

La variable de portée est similaire à la variable d'itération dans une boucle `foreach`, à la différence qu'aucune itération réelle ne se produit dans une expression de requête. Quand la requête est exécutée, la variable de portée sert de référence à chaque élément consécutif dans `customers`. Comme le compilateur déduit le type de `cust`, vous n'avez pas à le spécifier explicitement. Des variables de portée supplémentaires peuvent être introduites par une clause `let`. Pour plus d'informations, consultez let, clause.

> **NOTE**
> Pour les sources de données non génériques telles que ArrayList, la variable de portée doit être explicitement typée. Pour plus d'informations, consultez Comment interroger un ArrayList avec LINQ (C#) et la clause from.

Filtrage

L'opération de requête la plus courante est probablement l'application d'un filtre sous forme d'expression booléenne. Du fait du filtre, la requête retourne uniquement les éléments pour lesquels l'expression a la valeur true. Le résultat est produit à l'aide de la clause `where`. En effet, le filtre spécifie les éléments à exclure de la séquence source. Dans l'exemple suivant, seuls les clients (`customers`) qui ont une adresse à Londres sont retournés.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Vous pouvez utiliser les opérateurs logiques C# `AND` et `OR` habituels pour appliquer le nombre d'expressions de filtre nécessaire dans la clause `where`. Par exemple, pour retourner uniquement les clients situés à « Londres » ET (`AND`) dont le nom est « Devon », vous devez écrire le code suivant :

```
where cust.City == "London" && cust.Name == "Devon"
```

Pour retourner les clients situés à Londres ou à Paris, vous devez écrire le code suivant :

```
where cust.City == "London" || cust.City == "Paris"
```

Pour plus d'informations, consultez where, clause.

Classement

Il est souvent utile de trier les données retournées. La clause `orderby` permet de trier les éléments de la séquence retournée en fonction du comparateur par défaut pour le type qui est trié. Par exemple, la requête suivante peut être étendue pour trier les résultats selon la propriété `Name`. Comme `Name` est une chaîne, le comparateur par défaut effectue un tri alphabétique de A à Z.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Pour trier les résultats dans l'ordre inverse, de Z à A, utilisez la clause `orderby…descending`.

Pour plus d'informations, consultez orderby, clause.

Regroupement

La clause `group` vous permet de grouper vos résultats selon une clé que vous spécifiez. Par exemple, vous pouvez spécifier un regroupement des résultats par ville (`City`) pour que tous les clients de Londres ou Paris soient dans des groupes individuels. Dans ce cas, utilisez la clé `cust.City`.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Quand vous terminez une requête avec une clause `group`, les résultats sont présentés sous la forme d'une liste de listes. Chaque élément de la liste est un objet qui a un membre `Key` et une liste d'éléments qui sont regroupés sous cette clé. Quand vous itérez une requête qui produit une séquence de groupes, vous devez utiliser une boucle `foreach` imbriquée. La boucle externe itère chaque groupe et la boucle interne itère les membres de chaque groupe.

Si vous devez faire référence aux résultats d'une opération de regroupement, utilisez le mot clé `into` pour créer un identificateur susceptible d'être interrogé ultérieurement. La requête suivante retourne uniquement les groupes qui contiennent plus de deux clients :

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Pour plus d'informations, consultez Group, clause.

Jonction

Les opérations de jointure créent des associations entre les séquences qui ne sont pas explicitement modélisées dans les sources de données. Par exemple, effectuez une jointure pour rechercher tous les clients et distributeurs qui se trouvent au même endroit. Dans LINQ, la `join` clause fonctionne toujours sur les collections d'objets plutôt que sur les tables de base de données directement.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

Dans LINQ, il n'est pas nécessaire d'utiliser `join` aussi souvent que dans SQL, car les clés étrangères dans LINQ sont représentées dans le modèle objet sous la forme de propriétés qui contiennent une collection d'éléments. Par exemple, un objet `Customer` contient une collection d'objets `Order`. Au lieu d'effectuer une jointure, accédez aux commandes en utilisant la notation par points :

```
from order in Customer.Orders...
```

Pour plus d'informations, consultez join, clause.

Sélection (projections)

La clause `select` produit les résultats de la requête et spécifie la « forme » ou le type de chaque élément retourné. Par exemple, spécifiez si les résultats doivent contenir des objets `Customer` complets, un seul membre, un sous-ensemble de membres ou un type de résultat complètement différent basé sur un calcul ou une création d'objet. Quand la clause `select` produit autre chose qu'une copie de l'élément source, l'opération est appelée *projection*. L'utilisation de projections pour transformer des données est une fonctionnalité puissante des expressions de requête LINQ. Pour plus d'informations, consultez Transformations de données avec LINQ (C#) et select, clause.

Voir aussi

- Expressions de requête LINQ

- Procédure pas à pas : écriture de requêtes en C#
- Mots clés de requête (LINQ)
- Types anonymes

Transformations de données avec LINQ (C#)

02/11/2020 • 9 minutes to read • Edit Online

LINQ (Language-Integrated Query) n'est pas seulement la récupération des données. C'est également un outil puissant pour la transformation de données. À l'aide d'une requête LINQ, vous pouvez utiliser une séquence source comme entrée et la modifier de nombreuses façons pour créer une séquence de sortie. Vous pouvez modifier la séquence elle-même sans modifier les éléments eux-mêmes en les triant et en les regroupant. Mais la fonctionnalité la plus puissante des requêtes LINQ est peut-être la possibilité de créer des types. Cette opération est effectuée dans la clause select. Par exemple, il est possible de réaliser les tâches suivantes :

- Fusionner plusieurs séquences d'entrée en une séquence de sortie unique ayant un nouveau type.
- Créer des séquences de sortie dont les éléments se composent d'une seule propriété ou de plusieurs propriétés de chaque élément de la séquence source.
- Créer des séquences de sortie dont les éléments se composent des résultats des opérations effectuées sur les données sources.
- Créer des séquences de sortie dans un format différent. Par exemple, vous pouvez transformer des données de lignes ou de fichiers texte SQL en données XML.

Voici une liste non exhaustive d'exemples. Bien sûr, ces transformations peuvent être combinées de plusieurs façons dans la même requête. Par ailleurs, la séquence de sortie d'une requête peut être utilisée comme séquence d'entrée pour une nouvelle requête.

Combinaison de plusieurs entrées en une séquence de sortie

Vous pouvez utiliser une requête LINQ pour créer une séquence de sortie qui contient des éléments issus de plusieurs séquences d'entrée. L'exemple suivant montre comment combiner deux structures de données en mémoire, mais les mêmes principes peuvent être appliqués pour combiner des données provenant de sources XML, SQL ou DataSet. Prenons l'exemple des deux types de classe suivants :

```
class Student
{
    public string First { get; set; }
    public string Last {get; set;}
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

L'exemple suivant présente la requête :

```
class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
            new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                    where student.City == "Seattle"
                    select student.Last)
                    .Concat(from teacher in teachers
                            where teacher.City == "Seattle"
                            select teacher.Last);

        Console.WriteLine("The following students and teachers live in Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The following students and teachers live in Seattle:
    Omelchenko
    Beebe
 */
```

Pour plus d'informations, consultez join, clause et select, clause.

Sélection d'un sous-ensemble de chaque élément source

Pour sélectionner un sous-ensemble de chaque élément de la séquence source, deux méthodes principales

s'offrent à vous :

1. Pour sélectionner un seul membre de l'élément source, utilisez l'opérateur point. Dans l'exemple suivant, supposons qu'un objet `Customer` contienne plusieurs propriétés publiques comprenant une chaîne nommée `City`. Une fois exécutée, cette requête génère une séquence de sortie de chaînes.

```
var query = from cust in Customers
            select cust.City;
```

2. Pour créer des éléments qui contiennent plusieurs propriétés de l'élément source, vous pouvez utiliser un initialiseur d'objet avec un objet nommé ou un type anonyme. L'exemple suivant montre comment utiliser un type anonyme pour encapsuler deux propriétés de chaque élément `Customer` :

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

Pour plus d'informations, consultez Initialiseurs d'objets et de collections et Types anonymes.

Transformation d'objets en mémoire en XML

Les requêtes LINQ facilitent la transformation des données entre les structures de données en mémoire, les bases de données SQL, les jeux de données ADO.NET et les flux ou documents XML. L'exemple suivant transforme des objets d'une structure de données en mémoire en éléments XML.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
                       new XElement("First", student.First),
                       new XElement("Last", student.Last),
                       new XElement("Scores", scores)
                    ) // end "student"
                ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Le code génère la sortie XML suivante :

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

Pour plus d'informations, consultez Création d'arborescences XML en C# (LINQ to XML).

Exécution d'opérations sur les éléments sources

Une séquence de sortie ne contient pas toujours des éléments ou des propriétés d'élément de la séquence source. La sortie peut être plutôt une séquence de valeurs calculée en utilisant les éléments sources comme arguments d'entrée.

La requête suivante prend une séquence de nombres représentant des rayons de cercles, calcule la zone pour chaque rayon et retourne une séquence de sortie contenant des chaînes mises en forme avec la zone calculée.

Chaque chaîne de la séquence de sortie est mise en forme à l'aide de l' interpolation de chaîne. Une chaîne interpolée aura une `$` devant le guillemet ouvrant de la chaîne, et les opérations peuvent être effectuées entre accolades placées à l'intérieur de la chaîne interpolée. Une fois ces opérations effectuées, les résultats sont concaténés.

> **NOTE**
>
> Les méthodes d'appel dans les expressions de requête ne sont pas prises en charge si la requête est traduite dans un autre domaine. Par exemple, vous ne pouvez pas appeler une méthode C# ordinaire dans LINQ to SQL, car SQL Server n'a pas de contexte pour cette méthode. Toutefois, vous pouvez mapper des procédures stockées à des méthodes et appeler celles-ci. Pour plus d'informations, consultez Procédures stockées.

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area for a circle with a radius of '1' = 3.14
    Area for a circle with a radius of '2' = 12.57
    Area for a circle with a radius of '3' = 28.27
*/
```

Voir aussi

- LINQ (Language-Integrated Query) (C#)
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML (C#)
- Expressions de requête LINQ
- clause SELECT

Relations des types dans des opérations de requête LINQ (C#)

02/11/2020 • 5 minutes to read • Edit Online

Pour écrire efficacement des requêtes, vous devez comprendre comment les types des variables dans une opération de requête complète sont liés les uns aux autres. Si vous comprenez ces relations, vous comprendrez plus facilement les exemples LINQ et les exemples de code dans la documentation. En outre, vous comprendrez ce qui se passe en arrière-plan lorsque des variables sont implicitement typées à l'aide de `var`.

Les opérations de requête LINQ sont fortement typées dans la source de données, dans la requête elle-même et dans l'exécution de la requête. Le type des variables dans la requête doit être compatible avec le type des éléments dans la source de données, ainsi qu'avec le type de la variable d'itération dans l'instruction `foreach`. Ce typage fort garantit l'interception des erreurs de type au moment de la compilation lorsqu'elles peuvent être corrigées avant que les utilisateurs ne les rencontrent.

Pour illustrer ces relations de types, la plupart des exemples qui suivent utilisent le typage explicite pour toutes les variables. Le dernier exemple montre comment les mêmes principes s'appliquent même si vous utilisez le typage implicite à l'aide de var.

Requêtes qui ne transforment pas les données sources

L'illustration suivante montre une opération de requête LINQ to Objects qui n'exécute aucune transformation sur les données. La source contient une séquence de chaînes et le résultat de la requête est également une séquence de chaînes.

1. L'argument de type de la source de données détermine le type de la variable de portée.
2. Le type de l'objet sélectionné détermine le type de la variable de requête. Ici, `name` est une chaîne. Par conséquent, la variable de requête est un `IEnumerable<string>`.
3. La variable de requête fait l'objet d'une itération dans l'instruction `foreach`. Comme la variable de requête est une séquence de chaînes, la variable d'itération est également une chaîne.

Requêtes qui transforment les données sources

L'illustration suivante montre une opération de requête LINQ to SQL qui effectue une transformation simple des données. La requête accepte une séquence d'objets `Customer` en entrée et sélectionne uniquement la propriété `Name` dans le résultat. Comme `Name` est une chaîne, la requête produit une séquence de chaînes en sortie.

```
Table<Customer> Customers = db.GetTable<Customers>();

IQueryable<string> custNameQuery =
                    from cust in Customers
                       where cust.City == "London"
                       select cust.Name;

foreach (string str in custNameQuery)
{
     Console.WriteLine(str);
}
```

1. L'argument de type de la source de données détermine le type de la variable de portée.
2. L'instruction `select` retourne la propriété `Name` à la place de l'objet `Customer` complet. Comme `Name` est une chaîne, l'argument de type de `custNameQuery` est `string`, et non pas `Customer`.
3. Comme `custNameQuery` est une séquence de chaînes, la variable d'itération de la boucle `foreach` doit également être de type `string`.

L'illustration suivante montre une transformation légèrement plus complexe. L'instruction `select` retourne un type anonyme qui capture seulement deux membres de l'objet `Customer` original.

```
Table<Customer> Customers = db.GetTable<Customers>();

var namePhoneQuery =
                 from cust in Customers
                   where cust.City == "London"
                   select new { name = cust.Name,
                        phone = cust.Phone };

foreach (var item in namePhoneQuery)
{
     Console.WriteLine(item);
}
```

1. L'argument de type de la source de données est toujours le type de la variable de portée dans la requête.
2. Comme l'instruction `select` génère un type anonyme, la variable de requête doit être implicitement typée à l'aide de `var`.
3. Comme le type de la variable de requête est implicite, la variable d'itération dans la boucle `foreach` doit également être implicite.

Laisser le compilateur déduire les informations de type

Vous devez comprendre les relations des types dans une opération de requête. Toutefois, vous avez la possibilité de laisser le compilateur faire tout le travail à votre place. Le mot clé var peut être utilisé pour toute variable locale dans une opération de requête. L'illustration suivante est similaire à l'exemple numéro 2 qui a été abordé précédemment. Toutefois, le compilateur fournit le type fort pour chaque variable dans l'opération de requête.

```
var Customers = db.GetTable<Customers>();

var custQuery = from cust in Customers
                where cust.City == "London"
                select cust;

foreach (var item in custQuery)
{
     Console.WriteLine(item);
}
```

Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.

Syntaxe de requête et syntaxe de méthode dans LINQ (C#)

02/11/2020 • 9 minutes to read • Edit Online

La plupart des requêtes dans la documentation LINQ (Language Integrated Query) d'introduction sont écrites à l'aide de la syntaxe de requête déclarative LINQ. Toutefois, la syntaxe de requête doit être traduite en appels de méthode pour le Common Langage Runtime (CLR) .NET lorsque le code est compilé. Ces appels de méthode appellent les opérateurs de requête standard, qui ont des noms tels que `Where`, `Select`, `GroupBy`, `Join`, `Max` et `Average`. Vous pouvez les appeler directement en utilisant la syntaxe de méthode à la place de la syntaxe de requête.

La syntaxe de requête et la syntaxe de méthode sont identiques sémantiquement, mais de nombreuses personnes trouvent la syntaxe de requête plus simple et plus facile à lire. Certaines requêtes doivent être exprimées en tant qu'appels de méthode. Par exemple, vous devez utiliser un appel de méthode pour exprimer une requête qui récupère le nombre d'éléments qui correspondent à une condition spécifiée. Vous devez également utiliser un appel de méthode pour une requête qui récupère dans une séquence source l'élément qui a la valeur maximale. En général, la documentation de référence des opérateurs de requête standard dans l'espace de noms System.Linq utilise la syntaxe de méthode. Par conséquent, même lorsque vous commencez à écrire des requêtes LINQ, il est utile de savoir comment utiliser la syntaxe de méthode dans les requêtes et dans les expressions de requête elles-mêmes.

Méthodes d'extension d'opérateur de requête standard

L'exemple suivant présente une *expression de requête* simple et la requête sémantiquement équivalente écrite en tant que *requête fondée sur une méthode*.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

La sortie des deux exemples est identique. Vous pouvez voir que le type de la variable de requête est le même dans les deux formes : IEnumerable<T>.

Pour comprendre la requête basée sur une méthode, examinons-la de plus près. Du côté droit de l'expression, remarquez que la clause `where` est maintenant exprimée comme une méthode d'instance sur l'objet `numbers`, qui, comme vous vous en rappelez, a un type `IEnumerable<int>`. Si vous êtes connaissez bien l'interface IEnumerable<T> générique, vous savez qu'elle n'a pas de méthode `Where`. Toutefois, si vous appelez la liste de saisie semi-automatique IntelliSense dans l'IDE de Visual Studio, vous ne verrez pas seulement une méthode `Where`, mais de nombreuses autres méthodes telles que `Select`, `SelectMany`, `Join` et `Orderby`. Il s'agit de tous les opérateurs de requête standard.

Même s'il semble que IEnumerable<T> a été redéfini pour inclure ces méthodes supplémentaires, ce n'est pas le cas, en réalité. Les opérateurs de requête standard sont implémentés comme un nouveau type de méthode,

appelé *méthodes d'extension*. Les méthodes d'extension « étendent » un type existant. Elles peuvent être appelées comme s'il s'agissait de méthodes d'instance sur le type. Les opérateurs de requête standard étendent IEnumerable<T>, si bien que vous pouvez écrire `numbers.Where(...)`.

Pour commencer à utiliser LINQ, il vous suffit de connaître les méthodes d'extension pour les mettre dans la portée de votre application à l'aide des directives appropriées `using`. Du point de vue de votre application, une méthode d'extension et une méthode d'instance normale sont identiques.

Pour plus d'informations sur les méthodes d'extension, consultez Méthodes d'extension. Pour plus d'informations sur les opérateurs de requête standard, consultez Présentation des opérateurs de requête standard (C#). Certains fournisseurs LINQ, tels que LINQ to SQL et LINQ to XML , implémentent leurs propres opérateurs de requête standard et des méthodes d'extension supplémentaires pour d'autres types en plus de IEnumerable<T> .

Expressions lambda

Dans l'exemple précédent, notez que l'expression conditionnelle (`num % 2 == 0`) est passée comme argument de ligne à la méthode `Where` : `Where(num => num % 2 == 0).` Cette expression inline est appelée expression lambda. C'est un moyen pratique d'écrire du code qui devrait autrement être écrit sous une forme plus fastidieuse, comme une méthode anonyme, un délégué générique ou une arborescence d'expressions. En C#, `=>` est l'opérateur lambda, qui se lit « conduit à ». Le `num` situé à gauche de l'opérateur est la variable d'entrée qui correspond à `num` dans l'expression de requête. Le compilateur peut déduire le type de `num`, car il sait que `numbers` est un type IEnumerable<T> générique. Le corps de l'expression lambda est identique à l'expression dans la syntaxe de requête ou dans toute autre expression ou instruction C#. Il peut inclure des appels de méthode et une autre logique complexe. La « valeur de retour » est simplement le résultat de l'expression.

Pour commencer à utiliser LINQ, il n'est pas nécessaire d'utiliser des expressions lambda de manière intensive. Toutefois, certaines requêtes peuvent être exprimées uniquement dans la syntaxe de méthode. Parmi elles, certaines nécessitent des expressions lambda. Une fois que vous vous familiariserez avec les expressions lambda, vous constaterez qu'il s'agit d'un outil puissant et flexible dans votre boîte à outils LINQ. Pour plus d'informations, consultez expressions lambda.

Composabilité des requêtes

Dans l'exemple de code précédent, notez que la méthode `OrderBy` est appelée à l'aide de l'opérateur point sur l'appel à `Where`. `Where` produit une séquence filtrée, puis `Orderby` agit sur cette séquence en la triant. Étant donné que les requêtes retournent un `IEnumerable`, vous les composez dans la syntaxe de méthode en chaînant les appels de méthode ensemble. Le compilateur effectue cette opération en arrière-plan lorsque vous écrivez des requêtes à l'aide de la syntaxe de requête. De plus, étant donné qu'une variable de requête ne stocke pas les résultats de la requête, vous pouvez la modifier ou l'utiliser à tout moment comme base d'une nouvelle requête, même après son exécution.

Fonctionnalités C# qui prennent en charge LINQ

02/11/2020 • 7 minutes to read • Edit Online

La section suivante présente de nouvelles constructions de langage qui sont apparues avec C# 3.0. Bien que ces nouvelles fonctionnalités soient toutes utilisées dans un certain degré avec des requêtes LINQ, elles ne sont pas limitées à LINQ et peuvent être utilisées dans n'importe quel contexte où elles sont utiles.

Expressions de requête

Les expressions de requête utilisent une syntaxe déclarative similaire à SQL ou XQuery pour interroger les collections IEnumerable. Au moment de la compilation, la syntaxe de requête est convertie en appels de méthode à l'implémentation d'un fournisseur LINQ des méthodes d'extension d'opérateur de requête standard. Les applications contrôlent les opérateurs de requête standard qui se trouvent dans la portée en spécifiant l'espace de noms approprié avec une directive `using`. L'expression de requête suivante accepte un tableau de chaînes, regroupe les chaînes qui commencent par le même caractère, puis trie ces groupes de chaînes.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Pour plus d'informations, consultez Expressions de requête LINQ.

Variables implicitement typées (var)

Au lieu de spécifier explicitement un type lorsque vous déclarez et initialisez une variable, vous pouvez utiliser le modificateur var pour indiquer au compilateur qu'il doit déduire et assigner le type, comme illustré ici :

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Les variables déclarées comme `var` sont tout aussi fortement typées en tant que variables dont le type est spécifié explicitement. `var` permet de créer des types anonymes, mais peut être utilisé uniquement pour les variables locales. Les tableaux peuvent également être déclarés avec un typage implicite.

Pour plus d'informations, consultez Variables locales implicitement typées.

Initialiseurs d'objets et de collections

Les initialiseurs d'objets et de collections permettent d'initialiser un objet sans appeler explicitement un constructeur pour cet objet. Les initialiseurs sont généralement utilisés dans les expressions de requête pour projeter la source de données en un nouveau type de données. En supposant une classe nommée `Customer` avec les propriétés publiques `Name` et `Phone`, l'initialiseur d'objet peut être utilisé, comme dans le code suivant :

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Si l'on reprend notre classe `Customer`, supposons qu'il existe une source de données appelée `IncomingOrders` et

que, pour chaque commande caractérisée par un grand `OrderSize`, nous souhaitons créer un nouveau `Customer` basé sur cette commande. Une requête LINQ peut être exécutée sur cette source de données et utiliser l'initialisation d'objets pour remplir une collection :

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

La source de données peut avoir plus de propriétés sous-jacentes que la classe `Customer`, par exemple, `OrderSize` ; cependant, avec l'initialisation d'objets, les données retournées par la requête sont moulées dans le type de données souhaité ; nous choisissons les données correspondant à notre classe. Par conséquent, nous disposons désormais d'un `IEnumerable` contenant les nouveaux `Customer` que nous voulions. La requête ci-dessus peut également s'écrire avec la syntaxe de méthode LINQ :

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name =
y.Name, Phone = y.Phone });
```

Pour plus d'informations, consultez les pages suivantes :

- Initialiseurs d'objets et de collections
- Syntaxe des expressions de requête pour les opérateurs de requête standard

Types anonymes

Un type anonyme est construit par le compilateur et le nom du type est uniquement disponible pour le compilateur. Les types anonymes permettent de regrouper facilement des propriétés de manière temporaire dans un résultat de requête, sans avoir à définir un type nommé distinct. Les types anonymes sont initialisés avec une nouvelle expression et un initialiseur d'objet, comme indiqué ici :

```
select new {name = cust.Name, phone = cust.Phone};
```

Pour plus d'informations, consultez types anonymes.

Méthodes d'extension

Une méthode d'extension est une méthode statique qui peut être associée à un type, de manière à être appelée comme une méthode d'instance de ce type. Cette fonctionnalité permet d'ajouter de nouvelles méthodes aux types existants sans les modifier. Les opérateurs de requête standard sont un ensemble de méthodes d'extension qui fournissent des fonctionnalités de requête LINQ pour tout type qui implémente IEnumerable<T> .

Pour plus d'informations, consultez Méthodes d'extension.

Expressions lambda

Une expression lambda est une fonction inline qui utilise l'opérateur => pour séparer les paramètres d'entrée du corps de la fonction, et qui peut être convertie au moment de la compilation en un délégué ou une arborescence d'expressions. Dans la programmation LINQ, vous rencontrerez des expressions lambda lorsque vous effectuerez des appels de méthode directs aux opérateurs de requête standard.

Pour plus d'informations, consultez les pages suivantes :

- Fonctions anonymes
- Expressions lambda

- Arborescences d'expressions (C#)

Voir aussi

- LINQ (Language-Integrated Query) (C#)

Procédure pas à pas : Écrire des requêtes en C# (LINQ)

02/11/2020 • 17 minutes to read • Edit Online

Cette procédure pas à pas présente les fonctionnalités du langage C# utilisées pour écrire des expressions de requête LINQ.

Créer un projet C#

> **NOTE**
> Les instructions suivantes s'appliquent à Visual Studio. Si vous utilisez un environnement de développement différent, créez un projet console avec une référence à System.Core.dll et une directive `using` pour l'espace de noms System.Linq.

Pour créer un projet dans Visual Studio

1. Démarrez Visual Studio.
2. Dans le menu principal, sélectionnez **Fichier**, **Nouveau**, **Projet**.

   La boîte de dialogue **Nouveau projet** s'affiche.
3. Développez **Installé**, **Modèles**, **Visual C#**, puis choisissez **Application console**.
4. Dans la zone de texte **Nom**, entrez un nom différent ou acceptez le nom par défaut, puis choisissez le bouton **OK**.

   Le nouveau projet s'affiche dans l'**Explorateur de solutions**.
5. Notez que votre projet a une référence à System.Core.dll et une directive `using` pour l'espace de noms System.Linq.

Créer une source de données en mémoire

La source de données pour les requêtes est une simple liste d'objets `Student`. Chaque enregistrement `Student` comporte un prénom, un nom et un tableau d'entiers représentant les notes d'examens en classe. Copiez ce code dans votre projet. Notez les caractéristiques suivantes :

- La classe `Student` se compose de propriétés implémentées automatiquement.
- Chaque étudiant de la liste est initialisé avec un initialiseur d'objet.
- La liste elle-même est initialisée avec un initialiseur de collection.

La totalité de cette structure de données est initialisée et instanciée sans appels explicites à un constructeur quelconque ni accès au membre explicite. Pour plus d'informations sur ces nouvelles fonctionnalités, consultez Propriétés implémentées automatiquement et Initialiseurs d'objets et de collection.

Pour ajouter la source de données

- Ajoutez la classe `Student` et la liste initialisée d'étudiants à la classe `Program` dans votre projet.

```
public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};
```

Pour ajouter un nouvel étudiant à la liste des étudiants

1. Ajoutez un nouvel élément `Student` à la liste `Students` et utilisez le nom et les notes d'examens de votre choix. Essayez d'entrer toutes les informations relatives au nouvel étudiant pour mieux apprendre la syntaxe de l'initialiseur d'objet.

Créer la requête

Pour créer une requête simple

- Dans la méthode `Main` de l'application, créez une requête simple qui, quand elle est exécutée, produit une liste de tous les étudiants dont la note pour le premier test est supérieure à 90. Notez que, puisque l'objet `Student` entier est sélectionné, le type de la requête est `IEnumerable<Student>`. Bien que le code puisse également utiliser un type implicite à l'aide du mot clé var, les types explicites sont utilisés pour illustrer clairement les résultats. (Pour plus d'informations sur `var` , consultez variables locales implicitement typées.)

  Notez également que la variable de portée de la requête, `student`, sert de référence à chaque `Student` dans la source, en fournissant l'accès au membre pour chaque objet.

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Exécuter la requête

Pour exécuter la requête

1. Écrivez maintenant la boucle `foreach` qui entraînera l'exécution de la requête. Notez les points suivants concernant le code :

   - Chaque élément de la séquence retournée est accessible via la variable d'itération dans la boucle `foreach`.

- Le type de cette variable est `Student` et le type de la variable de requête est compatible, `IEnumerable<Student>`.

2. Après avoir ajouté ce code, générez et exécutez l'application pour visualiser les résultats dans la fenêtre de **console**.

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Pour ajouter une autre condition de filtre

1. Vous pouvez associer plusieurs conditions booléennes dans la clause `where` pour affiner davantage une requête. Le code suivant ajoute une condition afin que la requête retourne les étudiants dont la première note est supérieure à 90 et dont la dernière est inférieure à 80. La clause `where` doit être semblable au code suivant.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Pour plus d'informations, consultez where, clause.

Modifier la requête

Pour classer les résultats

1. Il est plus facile d'analyser les résultats s'ils sont classés. Vous pouvez classer la séquence retournée selon tous les champs accessibles dans les éléments sources. Par exemple, la clause `orderby` suivante classe les résultats dans l'ordre alphabétique de A à Z d'après le nom de chaque étudiant. Ajoutez la clause `orderby` suivante à votre requête, juste après l'instruction `where` et avant l'instruction `select` :

```
orderby student.Last ascending
```

2. Modifiez à présent la clause `orderby` pour qu'elle classe les résultats dans l'ordre inverse d'après la note au premier examen, de la plus élevée à la plus faible.

```
orderby student.Scores[0] descending
```

3. Modifiez la chaîne de format `WriteLine` pour pouvoir consulter les notes :

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

Pour plus d'informations, consultez orderby, clause.

Pour regrouper les résultats

1. Le regroupement est une fonction puissante des expressions de requête. Une requête avec une clause group produit une séquence de groupes dont chaque groupe contient lui-même un élément `Key` et une séquence composée de tous les membres de ce groupe. La nouvelle requête suivante regroupe les étudiants en utilisant la première lettre de leur nom comme clé.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. Notez que le type de la requête a maintenant changé. Elle produit désormais une séquence de groupes qui ont un type `char` comme clé et une séquence d'objets `Student`. Étant donné que le type de la requête a changé, le code suivant modifie également la boucle d'exécution `foreach` :

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("   {0}, {1}",
                  student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

3. Exécutez l'application et affichez les résultats dans la fenêtre de **console**.

   Pour plus d'informations, consultez Group, clause.

Pour rendre les variables implicitement typées

1. Le codage explicite de `IEnumerables` de `IGroupings` peut rapidement s'avérer laborieux. Vous pouvez écrire la même requête et la même boucle `foreach` beaucoup plus facilement à l'aide de `var`. Le mot clé `var` ne modifie pas les types de vos objets ; il indique simplement au compilateur de déduire les types. Remplacez le type de `studentQuery` et la variable d'itération `group` par `var` et réexécutez la requête. Notez que, dans la boucle `foreach` interne, la variable d'itération est encore typée comme `Student` et que la requête fonctionne exactement comme précédemment. Remplacez la variable d'itération `s` par `var` et exécutez à nouveau la requête. Vous pouvez constater que les résultats obtenus sont exactement les mêmes.

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Pour plus d'informations sur var, consultez Variables locales implicitement typées.

Pour classer les groupes selon leur valeur de clé

1. Quand vous exécutez la requête précédente, vous remarquez que les groupes ne sont pas dans l'ordre alphabétique. Pour modifier cet ordre, vous devez fournir une clause `orderby` après la clause `group`. Toutefois, pour utiliser une clause `orderby`, vous avez d'abord besoin d'un identificateur servant de référence aux groupes créés par la clause `group`. Fournissez l'identificateur à l'aide du mot clé `into`, comme suit :

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Quand vous exécutez cette requête, vous voyez que les groupes sont désormais classés par ordre alphabétique.

Pour introduire un identificateur à l'aide de let

1. Vous pouvez utiliser le mot clé `let` pour introduire un identificateur pour tous les résultats d'expression dans l'expression de requête. Cet identificateur peut être pratique, comme dans l'exemple suivant, ou améliorer les performances en stockant les résultats d'une expression afin d'éviter qu'ils ne soient calculés plusieurs fois.

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

Pour plus d'informations, consultez let, clause.

Pour utiliser la syntaxe de méthode dans une expression de requête

1. Comme décrit dans Syntaxe de requête et syntaxe de méthode dans LINQ, certaines opérations de requête peuvent être exprimées uniquement à l'aide de la syntaxe de méthode. Le code suivant calcule la note totale de chaque `Student` dans la séquence source, puis appelle la méthode `Average()` sur les résultats de cette requête pour calculer la note moyenne de la classe.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

Pour transformer ou projeter la clause select

1. Il n'est pas rare qu'une requête produise une séquence dont les éléments diffèrent des éléments des séquences sources. Supprimez ou commentez votre requête et votre boucle d'exécution précédentes, et remplacez-les par le code suivant. Notez que la requête retourne une séquence de chaînes (pas `Students`) et que ce fait est répercuté dans la boucle `foreach`.

```
IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

2. Le code utilisé précédemment dans cette procédure pas à pas indique que la note moyenne de la classe est d'environ 334. Pour produire une séquence de `Students` dont la note totale est supérieure à la moyenne de classe, avec les `Student ID` correspondants, vous pouvez utiliser un type anonyme dans l'instruction `select` :

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Étapes suivantes

Une fois que vous êtes familiarisé avec les aspects de base de l'utilisation de requêtes en C#, vous êtes prêt à lire la documentation et les exemples pour le type spécifique de fournisseur LINQ qui vous intéresse :

LINQ to SQL

LINQ to DataSet

LINQ to XML (C#)

LINQ to Objects (C#)

Voir aussi

- LINQ (Language-Integrated Query) (C#)
- Expressions de requête LINQ

Présentation des opérateurs de requête standard (C#)

02/11/2020 • 6 minutes to read • Edit Online

Les *opérateurs de requête standard* sont les méthodes qui forment le modèle de requête LINQ. La plupart de ces méthodes fonctionnent sur des séquences, où une séquence est un objet dont le type implémente l'interface IEnumerable<T> ou l'interface IQueryable<T>. Les opérateurs de requête standard fournissent des fonctionnalités de requête, dont notamment le filtrage, la projection, l'agrégation, le tri, etc.

Il existe deux ensembles d'opérateurs de requête standard LINQ : un qui opère sur des objets de type IEnumerable<T> , un autre qui opère sur des objets de type IQueryable<T> . Les méthodes qui composent chaque ensemble sont des membres statiques des classes Enumerable et Queryable, respectivement. Elles sont définies en tant que *méthodes d'extension* du type sur lequel elles opèrent. Les méthodes d'extension peuvent être appelées à l'aide de la syntaxe de méthode statique ou de la syntaxe de méthode d'instance.

De plus, plusieurs méthodes d'opérateur de requête standard fonctionnent sur des types autres que ceux basés sur IEnumerable<T> ou IQueryable<T>. Le type Enumerable définit deux de ces méthodes qui fonctionnent toutes deux sur les objets de type IEnumerable. Ces méthodes, Cast<TResult>(IEnumerable) et OfType<TResult>(IEnumerable), vous permettent d'autoriser l'interrogation d'une collection non paramétrée, ou non générique, dans le modèle LINQ. Pour ce faire, ils créent une collection fortement typée d'objets. La classe Queryable définit deux méthodes similaires, Cast<TResult>(IQueryable) et OfType<TResult>(IQueryable), qui opèrent sur les objets de type Queryable.

Les opérateurs de requête standard diffèrent dans le déroulement de leur exécution, selon qu'ils retournent une valeur singleton ou une séquence de valeurs. Les méthodes qui retournent une valeur de singleton (par exemple, Average et Sum) s'exécutent immédiatement. Les méthodes qui retournent une séquence diffèrent l'exécution de la requête et retournent un objet énumérable.

Pour les méthodes qui opèrent sur des collections en mémoire, autrement dit, les méthodes qui étendent IEnumerable<T> , l'objet énumérable retourné capture les arguments passés à la méthode. Lorsque cet objet est énuméré, la logique de l'opérateur de requête est employée et les résultats de requête sont retournés.

En revanche, les méthodes qui étendent IQueryable<T> n'implémentent aucun comportement d'interrogation. Ils créent une arborescence d'expressions qui représente la requête à exécuter. Le traitement des requêtes est géré par l'objet IQueryable<T> source.

Les appels aux méthodes de requête peuvent être chaînés dans une requête unique, ce qui permet de rendre les requêtes arbitrairement complexes.

L'exemple de code suivant illustre l'utilisation des opérateurs de requête standard pour obtenir des informations sur une séquence.

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Syntaxe d'expression de requête

Certains des opérateurs de requête standard les plus fréquemment utilisés ont une syntaxe de mot clé C# et Visual Basic Language dédiée qui leur permet d'être appelés dans le cadre d'une *expression*de *requête* . Pour plus d'informations sur les opérateurs de requête standard qui ont des mots clés dédiés et sur leurs syntaxes correspondantes, consultez Syntaxe des expressions de requête pour les opérateurs de requête standard (C#).

Extension des opérateurs de requête standard

Vous pouvez augmenter l'ensemble d'opérateurs de requête standard en créant des méthodes spécifiques au domaine appropriées pour votre domaine ou technologie cible. Vous pouvez également remplacer les opérateurs de requête standard par vos propres implémentations qui fournissent des services supplémentaires, tels que l'évaluation à distance, la traduction des requêtes et l'optimisation. Pour obtenir un exemple, consultez AsEnumerable.

Sections connexes

Les liens suivants vous permettent d'accéder à des articles qui fournissent des informations supplémentaires sur les différents opérateurs de requête standard en fonction de leur fonctionnalité.

Tri des données (C#)

Opérations ensemblistes (C#)

Filtrage des données (C#)

Opérations de quantificateur (C#)

Opérations de projection (C#)

Partitionnement des données (C#)

Opérations de jointure (C#)

Regroupement de données (C#)

Opérations de génération (C#)

Opérations d'égalité (C#)

Opérations d'élément (C#)

Conversion des types de données (C#)

Opérations de concaténation (C#)

Opérations d'agrégation (C#)

Voir aussi

- Enumerable
- Queryable
- Introduction aux requêtes LINQ (C#)
- Syntaxe des expressions de requête pour les opérateurs de requête standard (C#)
- Classification des opérateurs de requête standard en fonction de leur mode d'exécution (C#)
- Méthodes d'extension

Syntaxe des expressions de requête pour les opérateurs de requête standard (C#)

02/11/2020 • 2 minutes to read • Edit Online

Certains des opérateurs de requête standard les plus courants ont une syntaxe de mots clés C# dédiée qui permet de les appeler dans le cadre d'une *expression de requête*. Une expression de requête est une façon différente et plus lisible d'exprimer une requête que son équivalent *fondé sur une méthode*. Les clauses d'expression de requête sont traduites en appels aux méthodes de requête lors de la compilation.

Tableau de syntaxe des expressions de requête

Le tableau ci-dessous répertorie les opérateurs de requête standard qui comportent des clauses d'expression de requête équivalentes.

MÉTHODE	SYNTAXE D'EXPRESSION DE REQUÊTE C#
Cast	Utilisez une variable de portée explicitement typée, par exemple : `from int i in numbers` (Pour plus d'informations, consultez from, clause.)

GroupBy	`group … by`
	- ou -
	`group … by … into …`
	(Pour plus d'informations, consultez group, clause.)

GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	`join … in … on … equals … into …` (Pour plus d'informations, consultez join, clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Pour plus d'informations, consultez join, clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Pour plus d'informations, consultez orderby, clause.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Pour plus d'informations, consultez orderby, clause.)
Select	`select` (Pour plus d'informations, consultez select, clause.)

MÉTHODE	SYNTAXE D'EXPRESSION DE REQUÊTE C#
SelectMany	Plusieurs clauses `from`. (Pour plus d'informations, consultez from, clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Pour plus d'informations, consultez orderby, clause.)
ThenByDescending<TSource,TKey> (IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Pour plus d'informations, consultez orderby, clause.)
Where	`where` (Pour plus d'informations, consultez Where, clause.)

Voir aussi

- Enumerable
- Queryable
- Présentation des opérateurs de requête standard (C#)
- Classification des opérateurs de requête standard en fonction de leur mode d'exécution (C#)

Classification des opérateurs de requête standard en fonction de leur mode d'exécution (C#)

02/11/2020 • 6 minutes to read • Edit Online

Les implémentations LINQ to Objects des méthodes d'opérateur de requête standard s'exécutent de deux manières principales : immédiate ou différée. En outre, les opérateurs de requête qui utilisent l'exécution différée peuvent être divisés en deux catégories : ceux prenant en charge la diffusion en continu et ceux ne la prenant pas en charge. Le fait de connaître le mode d'exécution des différents opérateurs de requête peut vous aider à comprendre les résultats obtenus à partir d'une requête donnée. Ceci est particulièrement vrai si la source de données change ou si vous générez une requête sur une autre requête. Cette rubrique classe les opérateurs de requête standard selon leur mode d'exécution.

Modes d'exécution

Immédiat

L'exécution immédiate signifie que la source de données est lue et que l'opération est effectuée au point où la requête est déclarée dans le code. Tous les opérateurs de requête standard qui retournent un résultat unique et non énumérable s'exécutent immédiatement.

Différé

L'exécution différée signifie que l'opération n'est pas effectuée au point où la requête est déclarée dans le code. L'opération est effectuée uniquement quand la variable de requête est énumérée, par exemple à l'aide d'une instruction `foreach`. Cela signifie que les résultats de l'exécution de la requête dépendent du contenu de la source de données au moment de l'exécution de la requête, et non au moment de sa définition. Si la variable de requête est énumérée plusieurs fois, les résultats peuvent différer chaque fois. Presque tous les opérateurs de requête standard dont le type de retour est IEnumerable<T> ou IOrderedEnumerable<TElement> s'exécutent de manière différée.

Les opérateurs de requête qui utilisent l'exécution différée peuvent également être classés comme prenant en charge la diffusion en continu ou non.

Diffusion en continu

Les opérateurs prenant en charge la diffusion en continu n'ont pas à lire toutes les données sources avant de générer des éléments. Au moment de l'exécution, un opérateur prenant en charge la diffusion en continu effectue son opération sur chaque élément source à mesure qu'il est lu et génère l'élément si nécessaire. Ce type d'opérateur continue à lire des éléments source jusqu'à ce qu'un élément de résultat puisse être produit. Cela signifie que plusieurs éléments source peuvent être lus pour produire un élément de résultat.

Sans diffusion en continu

Les opérateurs ne prenant pas en charge la diffusion en continu doivent lire toutes les données source avant de générer un élément de résultat. Les opérations telles que le tri ou le regroupement font partie de cette catégorie. Au moment de l'exécution, les opérateurs de requête ne prenant pas en charge la diffusion en continu lisent toutes les données source, les placent dans une structure de données, puis effectuent l'opération et génèrent les éléments de résultat.

Tableau de classification

Le tableau suivant classe chaque méthode d'opérateur de requête standard en fonction de son mode d'exécution.

NOTE

Si un opérateur est présent dans deux colonnes, deux séquences d'entrée sont impliquées dans l'opération et chacune d'elles est évaluée différemment. Dans ces cas, la première séquence dans la liste de paramètres est toujours évaluée de façon différée, avec diffusion en continu.

OPÉRATEUR DE REQUÊTE STANDARD	TYPE DE RETOUR	EXÉCUTION IMMÉDIATE	EXÉCUTION DIFFÉRÉE AVEC DIFFUSION EN CONTINU	EXÉCUTION DIFFÉRÉE SANS DIFFUSION EN CONTINU
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valeur numérique unique	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X

OPÉRATEUR DE REQUÊTE STANDARD	TYPE DE RETOUR	EXÉCUTION IMMÉDIATE	EXÉCUTION DIFFÉRÉE AVEC DIFFUSION EN CONTINU	EXÉCUTION DIFFÉRÉE SANS DIFFUSION EN CONTINU
Last	TSource	X		
LastOrDefault	TSource	X		
LongCount	Int64	X		
Max	Valeur numérique unique, TSource ou TResult	X		
Min	Valeur numérique unique, TSource ou TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Valeur numérique unique	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	

OPÉRATEUR DE REQUÊTE STANDARD	TYPE DE RETOUR	EXÉCUTION IMMÉDIATE	EXÉCUTION DIFFÉRÉE AVEC DIFFUSION EN CONTINU	EXÉCUTION DIFFÉRÉE SANS DIFFUSION EN CONTINU
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Tableau TSource	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

Voir aussi

- Enumerable
- Présentation des opérateurs de requête standard (C#)
- Syntaxe des expressions de requête pour les opérateurs de requête standard (C#)
- LINQ to Objects (C#)

Tri des données (C#)

02/11/2020 • 4 minutes to read • Edit Online

Une opération de tri ordonne les éléments d'une séquence en fonction d'un ou de plusieurs attributs. Le premier critère de tri effectue un tri principal sur les éléments. En spécifiant un deuxième critère de tri, vous pouvez trier les éléments dans chaque groupe de tri principal.

L'illustration suivante montre les résultats d'une opération de tri par ordre alphabétique sur une séquence de caractères :

Les méthodes d'opérateurs de requête standard qui trient les données sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
OrderBy	Trie les valeurs dans l'ordre croissant.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Trie les valeurs dans l'ordre décroissant.	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Effectue un tri secondaire dans l'ordre croissant.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Effectue un tri secondaire dans l'ordre décroissant.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Inverser	Inverse l'ordre des éléments dans une collection.	Non applicable.	Enumerable.Reverse Queryable.Reverse

Exemples de syntaxe d'expression de requête

Exemples de tri principal

Tri principal croissant

L'exemple suivant montre comment utiliser la clause `orderby` dans une requête LINQ pour trier les chaînes dans un tableau par longueur de chaîne, dans l'ordre croissant.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/
```

Tri principal décroissant

L'exemple suivant montre comment utiliser la clause `orderby descending` dans une requête LINQ pour trier les chaînes selon leur première lettre, dans l'ordre décroissant.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    quick
    jumps
    fox
    brown
*/
```

Exemples de tri secondaire

Tri secondaire croissant

L'exemple suivant montre comment utiliser la clause `orderby` dans une requête LINQ pour effectuer un tri principal et un tri secondaire des chaînes dans un tableau. Les chaînes sont triées en premier lieu selon leur longueur, puis secondairement en fonction de la première lettre de la chaîne, chaque fois dans l'ordre croissant.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/
```

Tri secondaire décroissant

L'exemple suivant montre comment utiliser la clause `orderby descending` dans une requête LINQ pour effectuer un tri principal dans l'ordre croissant et un tri secondaire dans l'ordre décroissant. Les chaînes sont triées en premier lieu selon leur longueur, puis secondairement en fonction de la première lettre de la chaîne.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- orderby, clause
- Classer les résultats d'une clause join
- Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Opérations ensemblistes (C#)

02/11/2020 • 3 minutes to read • Edit Online

Les opérations ensemblistes dans LINQ font référence à des opérations de requête qui génèrent un jeu de résultats basé sur la présence ou l'absence d'éléments équivalents dans la même collection (ou le même ensemble) ou dans une collection distincte (ou un ensemble distinct).

Les méthodes d'opérateur de requête standard qui effectuent des opérations ensemblistes sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Distinct	Supprime les valeurs en double d'une collection.	Non applicable.	Enumerable.Distinct Queryable.Distinct
Except	Retourne la différence ensembliste, à savoir les éléments d'une collection qui n'apparaissent pas dans une seconde collection.	Non applicable.	Enumerable.Except Queryable.Except
Intersect	Retourne l'intersection ensembliste, à savoir les éléments qui apparaissent dans chacune des deux collections.	Non applicable.	Enumerable.Intersect Queryable.Intersect
Union	Retourne l'union ensembliste, à savoir les éléments uniques qui apparaissent dans l'une ou l'autre des deux collections.	Non applicable.	Enumerable.Union Queryable.Union

Comparaison d'opérations ensemblistes

Distinct

L'exemple suivant illustre le comportement de la Enumerable.Distinct méthode sur une séquence de caractères. La séquence retournée contient les éléments uniques de la séquence d'entrée.

```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */
```

Except

L'exemple suivant illustre le comportement de Enumerable.Except . La séquence retournée contient uniquement les éléments de la première séquence d'entrée qui ne figurent pas dans la seconde séquence d'entrée.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

Intersect

L'exemple suivant illustre le comportement de Enumerable.Intersect . La séquence retournée contient les éléments qui sont communs aux deux séquences d'entrée.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

Union

L'exemple suivant illustre une opération d'Union sur deux séquences de caractères. La séquence retournée contient les éléments uniques des deux séquences d'entrée.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Comment combiner et comparer des collections de chaînes (LINQ) (C#)
- Guide pratique pour rechercher la différence définie entre deux listes (LINQ) (C#)

Filtrage des données (C#)

02/11/2020 • 2 minutes to read • Edit Online

Le filtrage fait référence à l'opération de restriction du jeu de résultats pour que celui-ci contienne uniquement les éléments qui répondent à une condition spécifiée. Cette opération est également appelée « sélection ».

L'illustration suivante montre les résultats du filtrage d'une séquence de caractères. Le prédicat de l'opération de filtrage spécifie que le caractère doit être « A ».

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de sélection sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
OfType	Sélectionne des valeurs, en fonction de leur capacité à être castées en un type spécifié.	Non applicable.	Enumerable.OfType Queryable.OfType
Where	Sélectionne les valeurs qui sont basées sur une fonction de prédicat.	`where`	Enumerable.Where Queryable.Where

Exemple de syntaxe d'expression de requête

L'exemple suivant utilise la clause `where` pour filtrer les chaînes d'un tableau qui ont une longueur spécifique.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- clause WHERE
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution
- Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ) (C#)
- Comment rechercher des fichiers avec un attribut ou un nom spécifié (C#)
- Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Opérations de quantificateur (C#)

02/11/2020 • 4 minutes to read • Edit Online

Les opérations de quantificateur retournent une valeur Boolean qui indique si certains ou tous les éléments d'une séquence remplissent une condition.

L'illustration suivante représente deux opérations de quantificateur différentes sur deux séquences sources différentes. La première opération demande si un ou plusieurs des éléments sont le caractère 'A' et le résultat est `true`. La deuxième opération demande si tous les éléments sont le caractère 'A' et le résultat est `true`.

Les méthodes d'opérateur de requête standard qui effectuent des opérations de quantificateur sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Tous	Détermine si tous les éléments d'une séquence remplissent une condition.	Non applicable.	Enumerable.All Queryable.All
Quelconque	Détermine si certains éléments d'une séquence remplissent une condition.	Non applicable.	Enumerable.Any Queryable.Any
Contient	Détermine si une séquence contient un élément spécifié.	Non applicable.	Enumerable.Contains Queryable.Contains

Exemples de syntaxe d'expression de requête

Tous

L'exemple suivant utilise `All` pour vérifier que toutes les chaînes ont une longueur spécifique.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length == 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}
```

Quelconque

L'exemple suivant utilise `Any` pour vérifier que toutes les chaînes commencent par « o ».

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item => item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}
```

Contient

L'exemple suivant utilise `Contains` pour vérifier qu'un tableau a un élément spécifique.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution
- Comment interroger des phrases qui contiennent un ensemble de mots spécifié (LINQ) (C#)

Opérations de projection (C#)

02/11/2020 • 5 minutes to read • Edit Online

La projection désigne l'opération de transformation d'un objet en une nouvelle forme qui se compose souvent uniquement des propriétés à utiliser ensuite. À l'aide de la projection, vous pouvez créer un nouveau type qui est généré à partir de chaque objet. Vous pouvez projeter une propriété et effectuer une fonction mathématique sur celle-ci. Vous pouvez également projeter l'objet d'origine sans le modifier.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de projection sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Sélectionnez	Projette les valeurs qui sont basées sur une fonction de transformation.	`select`	Enumerable.Select Queryable.Select
SelectMany	Projette les séquences de valeurs qui sont basées sur une fonction de transformation, puis les aplatit en une seule séquence.	Utilisation de plusieurs clauses `from`	Enumerable.SelectMany Queryable.SelectMany

Exemples de syntaxe d'expression de requête

Sélectionnez

L'exemple suivant utilise la clause `select` pour projeter la première lettre de chaque chaîne dans une liste de chaînes.

```
List<string> words = new List<string>() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

L'exemple suivant utilise plusieurs clauses `from` pour projeter tous les mots de chaque chaîne dans une liste de chaînes.

```
List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

Comparaison de Select et SelectMany

Les deux clauses `Select()` et `SelectMany()` retournent une valeur de résultat (ou des valeurs) à partir des valeurs sources. `Select()` retourne une seule valeur de résultat pour chaque valeur source. Le résultat global est donc une collection qui a le même nombre d'éléments que la collection source. En revanche, `SelectMany()` retourne un résultat global unique qui contient des sous-collections concaténées à partir de chaque valeur source. La fonction de transformation qui est passée comme argument à `SelectMany()` doit retourner une séquence énumérable de valeurs pour chaque valeur source. Ces séquences énumérables sont ensuite concaténées par `SelectMany()` pour créer une seule grande séquence.

Les deux illustrations suivantes montrent en quoi les actions de ces deux méthodes sont différentes d'un point de vue conceptuel. Dans chaque cas, supposons que la fonction (de transformation) du sélecteur sélectionne le tableau de fleurs (Flowers) à partir de chaque valeur source.

Cette illustration montre de quelle manière `Select()` retourne une collection qui a le même nombre d'éléments que la collection source.

Cette illustration montre de quelle façon `SelectMany()` concatène la séquence intermédiaire de tableaux en une seule valeur de résultat final qui contient chaque valeur de chaque tableau intermédiaire.

Exemple de code

L'exemple suivant compare le comportement de `Select()` et de `SelectMany()`. Le code crée un « bouquet » de fleurs en prenant les deux premiers éléments de chaque liste de noms de fleurs de la collection source. Dans cet exemple, la « valeur unique » que la fonction de transformation Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) utilise est elle-même une collection de valeurs. Cela nécessite d'utiliser la boucle `foreach` supplémentaire pour énumérer chaque chaîne dans chaque sous-séquence.

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>() {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" }},
        new Bouquet{ Flowers = new List<string> { "tulip", "rose", "orchid" }},
        new Bouquet{ Flowers = new List<string> { "gladiolis", "lily", "snapdragon", "aster", "protea" }},
        new Bouquet{ Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" }}
    };

    // *********** Select ***********
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ********* SelectMany *********
    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

       Results by using Select():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
```

```
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia

       Results by using SelectMany():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia
    */

}
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- clause SELECT
- Comment remplir des collections d'objets à partir de plusieurs sources (LINQ) (C#)
- Comment fractionner un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

Partitionnement des données (C#)

02/11/2020 • 2 minutes to read • Edit Online

Le partitionnement dans LINQ désigne l'opération consistant à diviser une séquence d'entrée en deux sections, sans réorganiser les éléments, puis à retourner l'une des sections.

L'illustration suivante montre les résultats de trois opérations de partitionnement différentes sur une séquence de caractères. La première opération retourne les trois premiers éléments de la séquence. La deuxième opération ignore les trois premiers éléments et retourne les éléments restants. La troisième opération ignore les deux premiers éléments de la séquence et retourne les trois éléments suivants.

Les méthodes d'opérateurs de requête standard qui partitionnent des séquences sont répertoriées dans la section suivante.

Opérateurs

NOM DE L'OPÉRATEUR	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Ignorer	Ignore les éléments jusqu'à une position spécifiée dans une séquence.	Non applicable.	Enumerable.Skip Queryable.Skip
SkipWhile	Ignore les éléments basés sur une fonction de prédicat jusqu'à un élément qui ne remplit pas la condition.	Non applicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Prend les éléments jusqu'à une position spécifiée dans une séquence.	Non applicable.	Enumerable.Take Queryable.Take
TakeWhile	Prend les éléments basés sur une fonction de prédicat jusqu'à un élément qui ne remplit pas la condition.	Non applicable.	Enumerable.TakeWhile Queryable.TakeWhile

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)

JoinOpérations (C#)

16/12/2020 • 6 minutes to read • Edit Online

Une *jointure* de deux sources de données est l'association des objets d'une source de données aux objets qui partagent un attribut commun dans une autre source de données.

JoinING est une opération importante dans les requêtes qui ciblent des sources de données dont les relations ne peuvent pas être suivies directement. En programmation orientée objet, cela peut correspondre à une corrélation entre objets qui n'est pas modélisée, par exemple la direction vers l'arrière dans une relation à sens unique. Voici un exemple de relation à sens unique : une classe Customer a une propriété de type City, alors que la classe City n'a aucune propriété correspondant à une collection d'objets Customer. Si vous avez une liste d'objets City et si vous souhaitez rechercher tous les clients de chaque ville, vous pouvez recourir à une opération de jointure.

Les méthodes de jointure fournies dans le framework LINQ sont Join et GroupJoin. Ces méthodes effectuent des équijointures, qui sont des jointures associant deux sources de données en fonction de l'égalité de leurs clés. (Pour comparaison, Transact-SQL prend en charge les opérateurs de jointure autres que « Equals », par exemple l'opérateur « inférieur à ».) En termes de base de données relationnelle, Join implémente une jointure interne, un type de jointure dans lequel seuls les objets qui ont une correspondance dans l'autre jeu de données sont retournés. La méthode GroupJoin n'a aucun équivalent direct dans le contexte des bases de données relationnelles, mais elle implémente un sur-ensemble de jointures internes et de jointures externes gauches. Une jointure externe gauche est une jointure qui retourne chaque élément de la source de données (gauche), même si elle n'a pas d'éléments corrélés dans l'autre source de données.

L'illustration suivante présente une vue conceptuelle de deux ensembles, ainsi que leurs éléments inclus dans une jointure interne ou une jointure externe gauche.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Join	Joins deux séquences basées sur des fonctions de sélection de clé et extrait des paires de valeurs.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Joindeux séquences basées sur des fonctions de sélection de clé et regroupent les correspondances résultantes pour chaque élément.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

Exemples de syntaxe d'expression de requête

Join

L'exemple suivant utilise la `join … in … on … equals …` clause pour joindre deux séquences en fonction d'une valeur spécifique :

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}
```

GroupJoin

L'exemple suivant utilise la `join … in … on … equals … into …` clause pour joindre deux séquences en fonction d'une valeur spécifique et regroupe les correspondances résultantes pour chaque élément :

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //      Tea
    // Group
    //    Apple
    //     Kiwi
    // Group
    //   Carrot
}
```

Voir aussi

- System.Linq
- Vue d'ensemble des opérateurs de requête standard (C#)
- Types anonymes
- Formulation Join des requêtes s et Cross-Product
- join, clause

- Joinà l'aide de clés composites
- Comment joindre du contenu issu de fichiers différents (LINQ) (C#)
- Classer les résultats d'une clause join
- Effectuer des opérations de jointure personnalisées
- Effectuer des jointures groupées
- Effectuer des jointures internes
- Effectuer des jointures externes gauches
- Comment remplir des collections d'objets à partir de plusieurs sources (LINQ) (C#)

Regroupement des données (C#)

02/11/2020 • 2 minutes to read • Edit Online

Le regroupement consiste à placer des données dans des groupes afin que les éléments de chaque groupe partagent un attribut commun.

L'illustration suivante montre les résultats du regroupement d'une séquence de caractères. La clé de chaque groupe est le caractère.

Les méthodes d'opérateurs de requête standard qui regroupent les éléments de données sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES

GroupBy	Regroupe les éléments qui partagent un attribut commun. Chaque groupe est représenté par un objet IGrouping<TKey,TElement>.	`group … by`
		- ou -
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

ToLookup	Insère des éléments dans un Lookup<TKey,TElement> (un dictionnaire de type un-à-plusieurs) basé sur une fonction de sélecteur de clés.	Non applicable.	Enumerable.ToLookup

Exemple de syntaxe d'expression de requête

L'exemple de code suivant utilise la clause `group by` pour regrouper des entiers dans une liste selon qu'ils sont pairs ou impairs.

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- group, clause
- Créer un groupe imbriqué
- Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)
- Regrouper les résultats d'une requête
- Effectuer une sous-requête sur une opération de regroupement
- Comment fractionner un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

Opérations de génération (C#)

02/11/2020 • 2 minutes to read • Edit Online

La génération fait référence à la création d'une nouvelle séquence de valeurs.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de génération sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
DefaultIfEmpty	Remplace une collection vide par une collection de singletons par défaut.	Non applicable.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Empty	Retourne une collection vide.	Non applicable.	Enumerable.Empty
Plage	Génère une collection qui contient une séquence de nombres.	Non applicable.	Enumerable.Range
Répéter	Génère une collection qui contient une valeur répétée.	Non applicable.	Enumerable.Repeat

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)

Opérations d'égalité (C#)

02/11/2020 • 2 minutes to read • Edit Online

Deux séquences dont les éléments correspondants sont égaux et qui ont le même nombre d'éléments sont considérées comme égales.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
SequenceEqual	Détermine si deux séquences sont égales en comparant les éléments par paires.	Non applicable.	Enumerable.SequenceEqual Queryable.SequenceEqual

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Comment comparer le contenu de deux dossiers (LINQ) (C#)

Opérations d'éléments (C#)

02/11/2020 • 2 minutes to read • Edit Online

Les opérations d'éléments retournent un élément unique et spécifique à partir d'une séquence.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations d'éléments sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
ElementAt	Retourne l'élément situé à un index spécifié dans une collection.	Non applicable.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Retourne l'élément situé à un index spécifié dans une collection ou une valeur par défaut si l'index est hors limites.	Non applicable.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	Retourne le premier élément d'une collection ou le premier élément qui remplit une condition.	Non applicable.	Enumerable.First Queryable.First
FirstOrDefault	Retourne le premier élément d'une collection ou le premier élément qui remplit une condition. Retourne une valeur par défaut s'il n'existe aucun élément répondant aux critères.	Non applicable.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Dernier	Retourne le dernier élément d'une collection ou le dernier élément qui remplit une condition.	Non applicable.	Enumerable.Last Queryable.Last
LastOrDefault	Retourne le dernier élément d'une collection ou le dernier élément qui remplit une condition. Retourne une valeur par défaut s'il n'existe aucun élément répondant aux critères.	Non applicable.	Enumerable.LastOrDefault Queryable.LastOrDefault

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Unique	Retourne le seul élément d'une collection ou le seul élément qui remplit une condition. Lève une InvalidOperationException s'il n'y a aucun élément ou s'il y a plusieurs éléments à retourner.	Non applicable.	Enumerable.Single Queryable.Single
SingleOrDefault	Retourne le seul élément d'une collection ou le seul élément qui remplit une condition. Retourne une valeur par défaut s'il n'y a aucun élément à retourner. Lève une InvalidOperationException s'il y a plusieurs éléments à retourner.	Non applicable.	Enumerable.SingleOrDefault Queryable.SingleOrDefault

Voir aussi

- System.Linq
- Vue d'ensemble des opérateurs de requête standard (C#)
- Comment interroger les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)

Conversion des types de données (C#)

02/11/2020 • 4 minutes to read • Edit Online

Les méthodes de conversion changent le type d'objets en entrée.

Les opérations de conversion dans les requêtes LINQ sont utiles dans diverses applications. En voici quelques exemples :

- La méthode Enumerable.AsEnumerable peut être utilisée pour masquer l'implémentation personnalisée d'un type d'un opérateur de requête standard.
- La méthode Enumerable.OfType peut être utilisée pour activer des collections non paramétrables pour l'interrogation LINQ.
- Les méthodes Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList et Enumerable.ToLookup peuvent être utilisées pour forcer l'exécution immédiate de la requête au lieu de la différer jusqu'à ce que la requête soit énumérée.

Méthodes

Le tableau suivant répertorie les méthodes d'opérateur de requête standard qui effectuent des conversions de types de données.

Les méthodes de conversion de ce tableau dont les noms commencent par "As" modifient le type statique de la collection source mais ne l'énumèrent pas. Les méthodes dont les noms commencent par "To" énumèrent la collection source et placent les éléments dans le type de collection correspondant.

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
AsEnumerable	Retourne l'entrée typée comme IEnumerable<T>.	Non applicable.	Enumerable.AsEnumerable
AsQueryable	Convertit un IEnumerable (générique) en IQueryable (générique).	Non applicable.	Queryable.AsQueryable
Caster	Effectue un cast des éléments d'une collection en un type spécifié.	Utilisez une variable de portée explicitement typée. Par exemple : `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Filtre les valeurs en fonction de leur capacité à faire l'objet d'un cast en un type spécifié.	Non applicable.	Enumerable.OfType Queryable.OfType
ToArray	Convertit une collection en un tableau. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToArray

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
ToDictionary	Place des éléments dans un Dictionary<TKey,TValue> basé sur une fonction de sélecteur de clés. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToDictionary
ToList	Convertit une collection en List<T>. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToList
ToLookup	Place des éléments dans un Lookup<TKey,TElement> (un dictionnaire de type un-à-plusieurs) basé sur une fonction de sélecteur de clés. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToLookup

Exemple de syntaxe d'expression de requête

L'exemple de code suivant utilise une variable de portée explicitement typée pour effectuer un cast d'un type en un sous-type avant d'accéder à un membre qui est disponible uniquement sur le sous-type.

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- from, clause
- Expressions de requête LINQ
- Comment interroger un ArrayList avec LINQ (C#)

Opérations de concaténation (C#)

02/11/2020 • 2 minutes to read • Edit Online

La concaténation fait référence à l'opération d'ajout d'une séquence à une autre.

L'illustration suivante représente une opération de concaténation de deux séquences de caractères.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de concaténation sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Concat	Concatène deux séquences pour en former une seule.	Non applicable.	Enumerable.Concat Queryable.Concat

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Comment combiner et comparer des collections de chaînes (LINQ) (C#)

Opérations d'agrégation (C#)

02/11/2020 • 2 minutes to read • Edit Online

Une opération d'agrégation calcule une valeur unique à partir d'une collection de valeurs. Par exemple, une opération d'agrégation peut être le calcul de la température quotidienne moyenne à partir des valeurs de température quotidiennes relevées sur un mois.

L'illustration suivante montre les résultats de deux opérations d'agrégation différentes sur une séquence de nombres. La première opération additionne les nombres. La deuxième opération retourne la valeur maximale dans la séquence.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations d'agrégation sont répertoriées dans la section suivante.

Méthodes

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Agrégat	Effectue une opération d'agrégation personnalisée sur les valeurs d'une collection.	Non applicable.	Enumerable.Aggregate Queryable.Aggregate
Average	Calcule la valeur moyenne d'une collection de valeurs.	Non applicable.	Enumerable.Average Queryable.Average
Count	Compte tous les éléments d'une collection, ou uniquement les éléments basés sur une fonction de prédicat.	Non applicable.	Enumerable.Count Queryable.Count
LongCount	Compte tous les éléments d'une grande collection, ou uniquement les éléments basés sur une fonction de prédicat.	Non applicable.	Enumerable.LongCount Queryable.LongCount
Max	Détermine la valeur maximale dans une collection.	Non applicable.	Enumerable.Max Queryable.Max

NOM DE LA MÉTHODE	DESCRIPTION	SYNTAXE D'EXPRESSION DE REQUÊTE C#	INFORMATIONS COMPLÉMENTAIRES
Min	Détermine la valeur minimale dans une collection.	Non applicable.	Enumerable.Min Queryable.Min
SUM	Calcule la somme des valeurs d'une collection.	Non applicable.	Enumerable.Sum Queryable.Sum

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Comment calculer des valeurs de colonnes dans un fichier texte CSV (LINQ) (C#)
- Comment interroger les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)
- Comment interroger le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)

LINQ to Objects (C#)

02/11/2020 • 3 minutes to read • Edit Online

« LINQ to Objects » fait référence à l'utilisation directe de requêtes LINQ avec n'importe quelle collection IEnumerable ou IEnumerable<T>, sans utiliser de fournisseur LINQ ou d'API intermédiaire comme LINQ to SQL ou LINQ to XML. Vous pouvez utiliser LINQ pour interroger des collections énumérables telles que List<T>, Array ou Dictionary<TKey,TValue>. La collection peut être définie par l'utilisateur ou être retournée par une API .NET.

Fondamentalement, LINQ to Objects représente une nouvelle approche des collections. Auparavant, vous deviez écrire des boucles `foreach` complexes pour spécifier comment récupérer les données d'une collection. Avec l'approche LINQ, vous écrivez du code déclaratif qui décrit ce que vous voulez récupérer.

De plus, les requêtes LINQ offrent trois avantages principaux par rapport aux boucles `foreach` classiques :

- Elles sont plus concises et lisibles, en particulier durant le filtrage de plusieurs conditions.
- Elles fournissent de puissantes fonctionnalités de filtrage, de classement et de regroupement avec un minimum de code d'application.
- Elles peuvent être portées vers d'autres sources de données avec peu ou pas de changements.

En général, plus l'opération que vous souhaitez effectuer sur les données est complexe, plus vous réaliserez d'avantages en utilisant LINQ au lieu des techniques d'itération traditionnelles.

Cette section a pour objectif de montrer l'approche basée sur LINQ avec quelques exemples sélectionnés. Elle n'est pas destinée à être exhaustive.

Dans cette section

LINQ et chaînes (C#)
Explique comment LINQ peut être utilisé pour interroger et transformer des chaînes et des collections de chaînes. Contient également des liens vers des articles qui illustrent ces principes.

LINQ et la réflexion (C#)
Contient un lien vers un exemple qui montre comment LINQ utilise la réflexion.

LINQ et répertoires de fichiers (C#)
Explique comment LINQ peut être utilisé pour interagir avec les systèmes de fichiers. Contient également des liens vers des articles qui illustrent ces concepts.

Comment interroger un ArrayList avec LINQ (C#)
Montre comment interroger une ArrayList en C#.

Comment ajouter des méthodes personnalisées pour les requêtes LINQ (C#)
Explique comment étendre l'ensemble des méthodes utilisables pour les requêtes LINQ en ajoutant des méthodes d'extension à l'interface IEnumerable<T>.

LINQ (Language-Integrated Query) (C#)
Fournit des liens vers des articles qui expliquent LINQ et fournissent des exemples de code qui effectuent des requêtes.

LINQ et chaînes (C#)

02/11/2020 • 5 minutes to read • Edit Online

LINQ peut être utilisé pour interroger et transformer des chaînes et des collections de chaînes. Il peut être particulièrement utile avec des données semi-structurées dans des fichiers texte. Les requêtes LINQ peuvent être combinées avec les fonctions de chaîne traditionnelles et les expressions régulières. Par exemple, vous pouvez utiliser la méthode String.Split ou Regex.Split pour créer un tableau de chaînes que vous pouvez ensuite interroger ou modifier à l'aide de LINQ. Vous pouvez utiliser la méthode Regex.IsMatch dans la clause `where` d'une requête LINQ. Et vous pouvez utiliser LINQ pour interroger ou modifier les résultats MatchCollection retournés par une expression régulière.

Vous pouvez également utiliser les techniques décrites dans cette section pour transformer des données texte semi-structurées en XML. Pour plus d'informations, consultez Comment générer du code XML à partir de fichiers CSV.

Les exemples de cette section se répartissent en deux catégories :

Interrogation d'un bloc de texte

Vous pouvez interroger, analyser et modifier des blocs de texte en les fractionnant en tableau requêtable de chaînes plus petites à l'aide de la méthode String.Split ou de la méthode Regex.Split. Vous pouvez fractionner le texte source en mots, en phrases, en paragraphes, en pages ou selon d'autres critères, puis effectuer des fractionnements supplémentaires s'ils sont nécessaires dans votre requête.

- Comment compter les occurrences d'un mot dans une chaîne (LINQ) (C#)
  Montre comment utiliser LINQ pour des interrogations simples sur du texte.
- Comment interroger des phrases qui contiennent un ensemble de mots spécifié (LINQ) (C#)

  Montre comment fractionner des fichiers texte avec des limites arbitraires et comment exécuter des requêtes sur chaque partie.
- Comment interroger des caractères dans une chaîne (LINQ) (C#)

  Montre qu'une chaîne est un type interrogeable.
- Comment combiner des requêtes LINQ avec des expressions régulières (C#)

  Montre comment utiliser des expressions régulières dans les requêtes LINQ pour des correspondances de modèle complexes sur des résultats de requête filtrés.

Interrogation de données semi-structurées au format texte

De nombreux types différents de fichiers texte sont constitués d'une série de lignes, souvent avec la même mise en forme, comme les fichiers délimités par des tabulations ou des virgules, ou des lignes de longueur fixe. Après avoir lu un tel fichier texte dans la mémoire, vous pouvez utiliser LINQ pour interroger et/ou modifier les lignes. Les requêtes LINQ simplifient également la combinaison de données provenant de plusieurs sources.

- Guide pratique pour rechercher la différence définie entre deux listes (LINQ) (C#)

  Montre comment rechercher toutes les chaînes qui sont présentes dans une liste mais pas dans l'autre.
- Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Montre comment trier les lignes de texte en fonction de n'importe quel mot ou champ.

- Comment réorganiser les champs d'un fichier délimité (LINQ) (C#)

  Montre comment réorganiser les champs dans une ligne d'un fichier .csv.

- Comment combiner et comparer des collections de chaînes (LINQ) (C#)

  Montre comment combiner des listes de chaînes de différentes façons.

- Comment remplir des collections d'objets à partir de plusieurs sources (LINQ) (C#)

  Montre comment créer des collections d'objets à l'aide de plusieurs fichiers texte comme sources de données.

- Comment joindre du contenu issu de fichiers différents (LINQ) (C#)

  Montre comment combiner des chaînes de deux listes en une seule chaîne en utilisant une clé en correspondance.

- Comment fractionner un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

  Montre comment créer des fichiers en utilisant un seul fichier comme source de données.

- Comment calculer des valeurs de colonnes dans un fichier texte CSV (LINQ) (C#)

  Montre comment effectuer des calculs mathématiques sur des données texte dans des fichiers .csv.

Voir aussi

- LINQ (Language-Integrated Query) (C#)
- Comment générer du code XML à partir de fichiers CSV

Comment compter les occurrences d'un mot dans une chaîne (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment utiliser une requête LINQ pour compter les occurrences d'un mot spécifié dans une chaîne. Notez que, pour effectuer le décompte, la méthode Split est d'abord appelée pour créer un tableau de mots. La méthode Split a un coût en matière de performances. Si la seule opération sur la chaîne consiste à compter les mots, il est préférable d'utiliser les méthodes Matches ou IndexOf. Toutefois, si les performances ne sont pas un facteur critique, ou si vous avez déjà fractionné la phrase pour effectuer d'autres types de requêtes sur elle, il est judicieux d'utiliser LINQ pour compter les mots ou les expressions.

Exemple

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
          @" have not been well integrated. Programmers work in C# or Visual Basic" +
          @" and also in SQL or XQuery. On the one side are concepts such as classes," +
          @" objects, fields, inheritance, and .NET APIs. On the other side" +
          @" are tables, columns, rows, nodes, and separate languages for dealing with" +
          @" them. Data types often require translation between the two worlds; there are" +
          @" different standard functions. Because the object world has no notion of query, a" +
          @" query can only be represented as a string without compile-time type checking or" +
          @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
          @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
StringSplitOptions.RemoveEmptyEntries);

        // Create the query.  Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                         where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                         select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\" were found.", wordCount, searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et

System.IO.

Voir aussi

- LINQ et chaînes (C#)

Comment interroger des phrases qui contiennent un ensemble de mots spécifié (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment rechercher dans un fichier texte les phrases qui contiennent des correspondances pour chaque ensemble de mots spécifié. Même si le tableau des termes de recherche est codé en dur dans cet exemple, il pourrait aussi être rempli dynamiquement lors de l'exécution. Dans cet exemple, la requête retourne les phrases qui contiennent les mots « Historically », « data » et « integrated ».

Exemple

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
        @"have not been well integrated. Programmers work in C# or Visual Basic " +
        @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
        @"objects, fields, inheritance, and .NET APIs. On the other side " +
        @"are tables, columns, rows, nodes, and separate languages for dealing with " +
        @"them. Data types often require translation between the two worlds; there are " +
        @"different standard functions. Because the object world has no notion of query, a " +
        @"query can only be represented as a string without compile-time type checking or " +
        @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
        @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                                   StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

La requête fractionne d'abord le texte en phrases, puis fractionne les phrases en tableaux de chaînes contenant

chacun des mots. Pour chacun de ces tableaux, la méthode Distinct supprime tous les mots en double, puis la requête effectue une opération Intersect sur le tableau de mots et le tableau `wordsToMatch`. Si le nombre de l'intersection est identique à celui du tableau `wordsToMatch`, cela signifie que tous les mots ont été trouvés et la phrase d'origine est donc retournée.

Dans l'appel à Split, les signes de ponctuation sont utilisés comme séparateurs pour être supprimés de la chaîne. Si vous ne l'avez pas fait, vous pourriez avoir, par exemple, la chaîne « Historically, », qui ne correspondrait pas au mot « Historically » du tableau `wordsToMatch`. Vous devrez peut-être utiliser des séparateurs supplémentaires, selon le type des signes de ponctuation qui se trouvent dans le texte source.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)

Comment interroger des caractères dans une chaîne (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

La classe String implémente l'interface IEnumerable<T> générique. De ce fait, il est possible d'interroger n'importe quelle chaîne comme une séquence de caractères. Toutefois, ceci n'est pas une utilisation courante de LINQ. Pour les opérations de critères spéciaux complexes, utilisez la classe Regex.

Exemple

L'exemple suivant interroge une chaîne pour déterminer le nombre de chiffres qu'elle contient. Notez que la requête est « réutilisée » après sa première exécution. Ceci est possible car la requête proprement dite ne stocke pas de résultats réels.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
          from ch in aString
          where Char.IsDigit(ch)
          select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- Comment combiner des requêtes LINQ avec des expressions régulières (C#)

Comment combiner des requêtes LINQ avec des expressions régulières (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment utiliser la classe Regex afin de créer une expression régulière pour les correspondances plus complexes des chaînes de texte. La requête LINQ facilite le filtrage des fichiers à parcourir à l'aide de l'expression régulière, et facilite également la personnalisation des résultats.

Exemple

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                                select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {
```

```
                Console.WriteLine("  " + v2);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // This method assumes that the application has discovery
    // permissions for all folders under the specified path.
    static IEnumerable<System.IO.FileInfo> GetFiles(string path)
    {
        if (!System.IO.Directory.Exists(path))
            throw new System.IO.DirectoryNotFoundException();

        string[] fileNames = null;
        List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

        fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
        foreach (string name in fileNames)
        {
            files.Add(new System.IO.FileInfo(name));
        }
        return files;
    }
}
```

Notez que vous pouvez également interroger l'objet MatchCollection qui est retourné par une recherche `RegEx`. Dans cet exemple, seule la valeur de chaque correspondance est générée dans les résultats. Toutefois, il est également possible d'utiliser LINQ pour effectuer toutes sortes de filtrage, de tri et de regroupement au sein de cette collection. Étant donné que MatchCollection est une collection IEnumerable non générique, vous devez déclarer explicitement le type de la variable de portée dans la requête.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- LINQ et répertoires de fichiers (C#)

Guide pratique pour rechercher la différence définie entre deux listes (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment utiliser LINQ pour comparer deux listes de chaînes et sortir les lignes qui sont présentes dans names1.txt, mais pas dans names2.txt.

Pour créer le fichier de données

1. Copiez names1.txt et names2.txt vers votre dossier de solution, comme indiqué dans Comment combiner et comparer des collections de chaînes (LINQ) (C#).

Exemple

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
          names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
     The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
     */
```

Certains types d'opérations de requête en C#, tels que Except, Distinct, Union et Concat, peuvent uniquement être exprimés dans une syntaxe basée sur une méthode.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)

Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

L'exemple suivant montre comment trier les lignes d'un texte structuré, telles que des valeurs séparées par des virgules, à l'aide d'un champ de la ligne. Le champ peut être spécifié de manière dynamique lors de l'exécution. Supposons que les champs du fichier scores.csv correspondent au numéro d'identification d'un étudiant, suivi d'une série de quatre notes d'examen.

Pour créer un fichier contenant des données

1. Copiez les données scores.csv à partir de la rubrique Comment joindre du contenu de fichiers non similaires (LINQ) (C#) et enregistrez-le dans votre dossier de solution.

Exemple

```
public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

Cet exemple montre également comment retourner une variable de requête à partir d'une méthode.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)

Comment réorganiser les champs d'un fichier délimité (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Un fichier de valeurs séparées par des virgules (CSV) est un fichier texte qui est souvent utilisé pour stocker des données de feuille de calcul ou autres données tabulaires qui sont représentées sous forme de lignes et de colonnes. En utilisant la méthode Split pour séparer les champs, il est très facile d'interroger et de manipuler des fichiers CSV à l'aide de LINQ. En fait, la même technique peut servir à réorganiser les sections de n'importe quelle ligne de texte structurée et ne se limite donc pas aux fichiers CSV.

Dans l'exemple suivant, supposons que les trois colonnes représentent le nom, le prénom et l'ID de l'étudiant. Les champs sont classés par ordre alphabétique, selon le nom des étudiants. La requête produit une séquence dans laquelle la colonne ID apparaît en premier, suivie d'une deuxième colonne qui comprend à la fois le prénom et le nom de l'étudiant. Les lignes sont réorganisées d'après le champ ID. Les résultats sont enregistrés dans un nouveau fichier et les données d'origine ne sont pas modifiées.

Pour créer le fichier de données

1. Copiez les lignes suivantes dans un fichier texte nommé spreadsheet1.csv. Enregistrez le fichier dans votre dossier de projet.

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Exemple

```
class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
 */
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- LINQ et répertoires de fichiers (C#)
- Comment générer du code XML à partir de fichiers CSV (C#)

Comment combiner et comparer des collections de chaînes (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment fusionner des fichiers qui contiennent des lignes de texte, puis comment trier les résultats. Il montre plus précisément comment effectuer une concaténation simple, une union et une intersection avec les deux ensembles de lignes de texte.

Pour configurer le projet et les fichiers texte

1. Copiez ces noms dans un fichier texte nommé names1.txt, puis enregistrez-le dans votre dossier de projet :

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copiez ces noms dans un fichier texte nommé names2.txt, puis enregistrez-le dans votre dossier de projet : Notez que les deux fichiers ont des noms en commun.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Exemple

```
class MergeStrings
    {
        static void Main(string[] args)
        {
            //Put text files in your solution folder
            string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
            string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

            //Simple concatenation and sort. Duplicates are preserved.
            IEnumerable<string> concatQuery =
                fileA.Concat(fileB).OrderBy(s => s);

            // Pass the query variable to another function for execution.
            OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved:");

            // Concatenate and remove duplicate names based on
            // default string comparer.
```

```
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
            fileA.Union(fileB).OrderBy(s => s);
        OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

        // Find the names that occur in both files (based on
        // default string comparer).
        IEnumerable<string> commonNamesQuery =
            fileA.Intersect(fileB);
        OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

        // Find the matching fields in each list. Merge the two
        // results by using Concat, and then
        // sort using the default string comparer.
        string nameMatch = "Garcia";

        IEnumerable<String> tempQuery1 =
            from name in fileA
            let n = name.Split(',')
            where n[0] == nameMatch
            select name;

        IEnumerable<string> tempQuery2 =
            from name2 in fileB
            let n2 = name2.Split(',')
            where n2[0] == nameMatch
            select name2;

        IEnumerable<string> nameMatchQuery =
            tempQuery1.Concat(tempQuery2).OrderBy(s => s);
        OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \"{nameMatch}\":");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    20 total names in list
```

```
        26 total names in list

        Union removes duplicate names:
        Aw, Kam Foo
        Bankov, Peter
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        16 total names in list

        Merge based on intersect:
        Bankov, Peter
        Holm, Michael
        Garcia, Hugo
        Beebe, Ann
        4 total names in list

        Concat based on partial name match "Garcia":
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        3 total names in list
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- LINQ et répertoires de fichiers (C#)

Comment remplir des collections d'objets à partir de plusieurs sources (LINQ) (C#)

02/11/2020 • 5 minutes to read • Edit Online

Cet exemple montre comment fusionner des données de différentes sources en une séquence de nouveaux types.

> **NOTE**
>
> N'essayez pas de joindre des données du système de fichiers ou des données en mémoire à des données qui se trouvent encore dans une base de données. Ces jointures interdomaines peuvent générer des résultats indéfinis, en raison des différentes façons par lesquelles les opérations de jointure peuvent être définies pour les requêtes de base de données et autres types de sources. En outre, une telle opération risque de provoquer une exception de mémoire insuffisante si la quantité de données dans la base de données est assez élevée. Pour joindre des données d'une base de données à des données en mémoire, appelez d'abord `ToList` ou `ToArray` sur la requête de base de données, puis effectuez la jointure sur la collection retournée.

Pour créer le fichier de données

Copiez les fichiers names.csv et scores.csv dans le dossier de votre projet, comme décrit dans Guide pratique pour joindre du contenu provenant de fichiers différents (LINQ) (C#).

Exemple

L'exemple suivant montre comment utiliser un type nommé `Student` pour stocker des données fusionnées de deux collections en mémoire de chaînes qui simulent des données de feuille de calcul au format .csv. La première collection de chaînes représente les noms et les ID des étudiants, et la deuxième collection représente l'ID d'étudiant (dans la première colonne) et quatre notes d'examen. L'ID est utilisé comme clé étrangère.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");
```

```
        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
        // creation of a list of ints for the ExamScores member. The first item
        // is skipped in the split string because it is the student ID,
        // not an exam score.
        IEnumerable<Student> queryNamesScores =
            from nameLine in names
            let splitName = nameLine.Split(',')
            from scoreLine in scores
            let splitScoreLine = scoreLine.Split(',')
            where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
            select new Student()
            {
                FirstName = splitName[0],
                LastName = splitName[1],
                ID = Convert.ToInt32(splitName[2]),
                ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries. This could be useful with
        // very large data files.
        List<Student> students = queryNamesScores.ToList();

        // Display each student's name and exam score average.
        foreach (var student in students)
        {
            Console.WriteLine("The average score of {0} {1} is {2}.",
                student.FirstName, student.LastName,
                student.ExamScores.Average());
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

Dans la clause select, un initialiseur d'objet est utilisé pour instancier chaque nouvel objet `Student` à l'aide des données des deux sources.

Si vous n'êtes pas obligé de stocker les résultats d'une requête, les types anonymes peuvent être plus pratiques que les types nommés. Les types nommés sont nécessaires si vous passez les résultats de requête en dehors de la méthode dans laquelle la requête est exécutée. L'exemple suivant exécute la même tâche que l'exemple précédent, mais utilise des types anonymes plutôt que des types nommés :

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}
```

Voir aussi

- LINQ et chaînes (C#)
- Initialiseurs d'objets et de collections
- Types anonymes

Comment fractionner un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment fusionner le contenu de deux fichiers, puis créer un ensemble de fichiers qui organisent les données d'une nouvelle façon.

Pour créer le fichier de données

1. Copiez ces noms dans un fichier texte nommé names1.txt, puis enregistrez-le dans votre dossier de projet :

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copiez ces noms dans un fichier texte nommé names2.txt, puis enregistrez-le dans votre dossier de projet. Notez que les deux fichiers ont certains noms en commun.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Exemple

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                         let n = name.Split(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;
```

```
        // Create a new file for each group that was created
        // Note that nested foreach loops are required to access
        // individual items with each group.
        foreach (var g in groupQuery)
        {
            // Create the new file name.
            string fileName = @"../../../testFile_" + g.Key + ".txt";

            // Output to display.
            Console.WriteLine(g.Key);

            // Write file.
            using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
            {
                foreach (var item in g)
                {
                    sw.WriteLine(item);
                    // Output to console for example purposes.
                    Console.WriteLine("   {0}", item);
                }
            }
        }
        // Keep console window open in debug mode.
        Console.WriteLine("Files have been written. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

Le programme écrit un fichier distinct pour chaque groupe, dans le même dossier que les fichiers de données.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- LINQ et répertoires de fichiers (C#)

Comment joindre du contenu issu de fichiers différents (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment joindre des données de deux fichiers CSV qui partagent une valeur commune utilisée comme clé correspondante. Cette technique peut être utile si vous devez combiner les données de deux feuilles de calcul, ou d'une feuille de calcul et d'un fichier qui a un autre format, dans un nouveau fichier. Vous pouvez modifier l'exemple pour qu'il fonctionne avec tout type de texte structuré.

Pour créer le fichier de données

1. Copiez les lignes suivantes dans un fichier nommé *scores.csv*, puis enregistrez-le dans votre dossier de projet. Le fichier représente les données de feuille de calcul. La colonne 1 est l'ID de l'étudiant et les colonnes 2 à 5 correspondent à des résultats d'examens.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Copiez les lignes suivantes dans un fichier nommé *names.csv*, puis enregistrez-le dans votre dossier de projet. Le fichier représente une feuille de calcul qui contient le nom, le prénom et l'ID de l'étudiant.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Exemple

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
```

```
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the scores to the student names by using ID as a
        // matching key.

        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Name:    Last[0],       First[1],  ID[2]
        //          Omelchenko,    Svetlana,  11
        // Score:   StudentID[0],  Exam1[1]   Exam2[2],  Exam3[3],  Exam4[4]
        //          111,           97,        92,        81,        60

        // This query joins two dissimilar spreadsheets based on common ID value.
        // Multiple from clauses are used instead of a join clause
        // in order to store results of id.Split.
        IEnumerable<string> scoreQuery1 =
            from name in names
            let nameFields = name.Split(',')
            from id in scores
            let scoreFields = id.Split(',')
            where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
            select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
                   + "," + scoreFields[3] + "," + scoreFields[4];

        // Pass a query variable to a method and execute it
        // in the method. The query itself is unchanged.
        OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

Voir aussi

- LINQ et chaînes (C#)

- LINQ et répertoires de fichiers (C#)

Comment calculer des valeurs de colonnes dans un fichier texte CSV (LINQ) (C#)

02/11/2020 • 4 minutes to read • Edit Online

Cet exemple montre comment effectuer des calculs d'agrégation tels que Sum, Average, Min et Max, avec les colonnes d'un fichier .csv. Les exemples de principes montrés ici sont applicables à d'autres types de texte structuré.

Pour créer le fichier source

1. Copiez les lignes suivantes dans un fichier nommé scores.csv, puis enregistrez-le dans votre dossier de projet. Supposons que la première colonne contienne un ID d'étudiant et que les colonnes suivantes contiennent les résultats de quatre examens.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Exemple

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");
```

```
        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at runtime.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery =
            from line in strs
            let elements = line.Split(',')
            select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
                  examNum, average, max, min);
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        IEnumerable<IEnumerable<int>> multiColQuery =
            from line in strs
            let elements = line.Split(',')
            let scores = elements.Skip(1)
            select (from str in scores
                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
```

```
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                          column + 1, average, max, min);
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
 */
```

La requête fonctionne à l'aide de la méthode Split pour convertir chaque ligne de texte en un tableau. Chaque élément du tableau représente une colonne. Enfin, le texte de chaque colonne est converti en sa représentation numérique. Si votre fichier est un fichier séparé par des tabulations, remplacez l'argument de la méthode `Split` par `\t`.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et chaînes (C#)
- LINQ et répertoires de fichiers (C#)

Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Les API de réflexion .NET peuvent être utilisées pour examiner les métadonnées dans un assembly .NET et créer des collections de types, des membres de type, des paramètres, etc., qui se trouvent dans cet assembly. Comme ces collections prennent en charge l'interface générique IEnumerable<T>, elles peuvent être interrogées à l'aide de LINQ.

L'exemple suivant montre comment utiliser LINQ avec la réflexion pour récupérer des métadonnées spécifiques concernant des méthodes qui correspondent à un critère de recherche spécifié. Ici, la requête va rechercher les noms de toutes les méthodes dans l'assembly qui retournent des types énumérables tels que des tableaux.

Exemple

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

L'exemple utilise la méthode Assembly.GetTypes pour retourner un tableau de types de l'assembly spécifié. Le filtre where est appliqué pour que seuls des types publics soient retournés. Pour chaque type public, une sous-requête est générée en utilisant le tableau MethodInfo qui est retourné à partir de l'appel Type.GetMethods. Ces résultats sont filtrés pour retourner uniquement les méthodes dont le type de retour est un tableau ou un type qui implémente IEnumerable<T>. Pour finir, ces résultats sont regroupés en utilisant le nom de type comme clé.

Voir aussi

- LINQ to Objects (C#)

Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Les API de réflexion .NET peuvent être utilisées pour examiner les métadonnées dans un assembly .NET et créer des collections de types, des membres de type, des paramètres, etc., qui se trouvent dans cet assembly. Comme ces collections prennent en charge l'interface générique IEnumerable<T>, elles peuvent être interrogées à l'aide de LINQ.

L'exemple suivant montre comment utiliser LINQ avec la réflexion pour récupérer des métadonnées spécifiques concernant des méthodes qui correspondent à un critère de recherche spécifié. Ici, la requête va rechercher les noms de toutes les méthodes dans l'assembly qui retournent des types énumérables tels que des tableaux.

Exemple

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

L'exemple utilise la méthode Assembly.GetTypes pour retourner un tableau de types de l'assembly spécifié. Le filtre where est appliqué pour que seuls des types publics soient retournés. Pour chaque type public, une sous-requête est générée en utilisant le tableau MethodInfo qui est retourné à partir de l'appel Type.GetMethods. Ces résultats sont filtrés pour retourner uniquement les méthodes dont le type de retour est un tableau ou un type qui implémente IEnumerable<T>. Pour finir, ces résultats sont regroupés en utilisant le nom de type comme clé.

Voir aussi

- LINQ to Objects (C#)

LINQ et répertoires de fichiers (C#)

02/11/2020 • 4 minutes to read • Edit Online

De nombreuses opérations du système de fichiers sont essentiellement des requêtes et sont donc bien adaptées à l'approche LINQ.

Les requêtes de cette section ne sont pas destructrices. Elles ne sont pas utilisées pour modifier le contenu des fichiers ou des dossiers d'origine. La règle suivie est que les requêtes ne doivent pas provoquer des effets secondaires. En règle générale, tout code (y compris les requêtes qui utilisent des opérateurs de création / mise à jour / suppression) qui modifie les données sources doit rester distinct du code qui interroge simplement les données.

Cette section contient les rubriques suivantes :

Comment rechercher des fichiers avec un attribut ou un nom spécifié (C#)
Montre comment rechercher des fichiers en examinant une ou plusieurs propriétés de son objet FileInfo.

Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)
Montre comment retourner des groupes d'objets FileInfo en fonction de leur extension de nom de fichier.

Comment interroger le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)
Montre comment retourner le nombre total d'octets dans tous les fichiers d'une arborescence de répertoires spécifiée.

Guide pratique pour comparer le contenu de deux dossiers (LINQ) (C#)
Montre comment retourner tous les fichiers qui sont présents dans deux dossiers spécifiés, ainsi que tous les fichiers qui sont présents dans un dossier mais pas dans l'autre.

Comment interroger les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)
Montre comment retourner le fichier le plus grand ou le plus petit, ou un nombre spécifié de fichiers, dans une arborescence de répertoires.

Comment interroger des fichiers dupliqués dans une arborescence de répertoires (LINQ) (C#)
Montre comment regrouper tous les noms de fichier qui se trouvent dans plusieurs emplacements d'une arborescence de répertoires spécifiée. Montre aussi comment effectuer des comparaisons plus complexes avec un comparateur personnalisé.

Comment interroger le contenu de fichiers dans un dossier (LINQ) (C#)
Montre comment itérer au sein des dossiers d'une arborescence, ouvrir chaque fichier et interroger le contenu du fichier.

Commentaires

La création d'une source de données représentant avec précision le contenu du système de fichiers et la gestion correcte des exceptions induisent une certaine complexité. Les exemples de cette section créent une collection d'instantanés d'objets FileInfo qui représente tous les fichiers d'un dossier racine spécifié et de tous ses sous-dossiers. L'état réel de chaque FileInfo peut changer dans le temps entre le début et la fin de l'exécution d'une requête. Par exemple, vous pouvez créer une liste d'objets FileInfo à utiliser comme source de données. Si vous essayez d'accéder à la propriété `Length` dans une requête, l'objet FileInfo tente d'accéder au système de fichiers pour mettre à jour la valeur de `Length`. Si le fichier n'existe plus, vous obtenez une exception FileNotFoundException dans votre requête, même si vous n'interrogez pas directement le système de fichiers. Certaines requêtes de cette section utilisent une méthode distincte qui consomme ces exceptions particulières

dans certains cas. Une autre option consiste à conserver votre source de données à jour dynamiquement en utilisant FileSystemWatcher.

Voir aussi

- LINQ to Objects (C#)

Comment rechercher des fichiers avec un attribut ou un nom spécifié (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment rechercher tous les fichiers ayant une extension de nom de fichier spécifiée (par exemple « .txt ») dans une arborescence de répertoires spécifiée. Il montre également comment retourner le fichier le plus récent ou le plus ancien dans l'arborescence en fonction de l'heure de création.

Exemple

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
             orderby file.CreationTime
             select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)

02/11/2020 • 4 minutes to read • Edit Online

Cet exemple montre comment utiliser LINQ pour effectuer des opérations de regroupement et de tri avancées sur des listes de fichiers ou de dossiers. Il montre également comment parcourir les résultats dans la fenêtre de console à l'aide des méthodes Skip et Take.

Exemple

La requête suivante montre comment regrouper le contenu d'une arborescence de répertoires spécifiée en fonction de l'extension de nom de fichier.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    //  and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                   IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;
```

```
        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var f in resultPage)
                {
                    Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

La sortie de ce programme peut être longue, en fonction des détails du système de fichiers local et de la valeur de `startFolder`. Pour permettre l'affichage de tous les résultats, cet exemple montre comment parcourir les résultats. Les mêmes techniques sont applicables aux applications Windows et web. Notez qu'étant donné que le code organise les éléments dans un groupe, une boucle `foreach` imbriquée est nécessaire. Il existe également une logique supplémentaire pour calculer la position actuelle dans la liste et pour permettre aux utilisateurs d'arrêter la pagination et de quitter le programme. Dans ce cas particulier, la requête de pagination est exécutée par rapport aux résultats de la requête d'origine mis en cache. Dans d'autres contextes, tels que LINQ to SQL, ce type de mise en cache n'est pas nécessaire.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Comment interroger le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre comment récupérer le nombre total d'octets utilisés par tous les fichiers d'un dossier spécifié, ainsi que par tous ses sous-dossiers.

Exemple

La méthode Sum ajoute les valeurs de tous les éléments sélectionnés dans la clause `select`. Vous pouvez facilement modifier cette requête pour récupérer le fichier le plus grand ou le plus petit de l'arborescence du répertoire spécifié en appelant la méthode Min ou Max plutôt que Sum.

```
class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}
```

Si vous devez seulement compter le nombre d'octets dans une arborescence de répertoires spécifiée, pour être plus efficace, ne créez pas de requête LINQ, car cela induit la surcharge liée à la création de la collection de listes comme source de données. Plus la requête est complexe, plus l'approche LINQ devient utile. C'est également valable lorsque vous devez exécuter plusieurs requêtes sur une même source de données.

La requête appelle une méthode distincte pour obtenir la longueur du fichier. Elle procède ainsi pour utiliser l'exception qui sera éventuellement levée si le fichier a été supprimé sur un autre thread après la création de l'objet FileInfo dans l'appel à `GetFiles`. Même si l'objet FileInfo a déjà été créé, l'exception peut être levée, car un objet FileInfo essaiera d'actualiser sa propriété Length avec la longueur la plus récente lors du premier accès à la propriété. En plaçant cette opération dans un bloc try-catch en dehors de la requête, le code respecte la règle qui consiste à éviter les opérations dans les requêtes qui peuvent avoir des effets secondaires. En règle générale, il faut

faire très attention lors de l'utilisation d'exceptions et s'assurer que l'application ne reste pas dans un état inconnu.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Comment comparer le contenu de deux dossiers (LINQ) (C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple montre trois façons de comparer deux listes de fichiers :

- En recherchant une valeur booléenne qui spécifie si les deux listes sont identiques
- En recherchant l'intersection pour récupérer les fichiers qui se trouvent dans les deux dossiers
- En recherchant la différence ensembliste pour récupérer les fichiers qui se trouvent dans un dossier, mais pas dans l'autre

> **NOTE**
>
> Les techniques présentées ici peuvent être adaptées pour comparer des séquences d'objets de tout type.

La classe `FileComparer` présentée ici montre comment utiliser une classe de comparateur personnalisée avec les opérateurs de requête standard. La classe n'est pas destinée à être utilisée dans des scénarios réels. Elle utilise simplement le nom et la longueur en octets de chaque fichier pour déterminer si les dossiers ont un contenu identique. Dans un scénario réel, vous devriez modifier ce comparateur pour effectuer une vérification d'égalité plus rigoureuse.

Exemple

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);
```

```
            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
            else
            {
                Console.WriteLine("The two folders are not the same");
            }

            // Find the common files. It produces a sequence and doesn't
            // execute until the foreach statement.
            var queryCommonFiles = list1.Intersect(list2, myFileCompare);

            if (queryCommonFiles.Any())
            {
                Console.WriteLine("The following files are in both folders:");
                foreach (var v in queryCommonFiles)
                {
                    Console.WriteLine(v.FullName); //shows which items end up in result list
                }
            }
            else
            {
                Console.WriteLine("There are no common files in the two folders.");
            }

            // Find the set difference between the two folders.
            // For this example we only check one way.
            var queryList1Only = (from file in list1
                                  select file).Except(list2, myFileCompare);

            Console.WriteLine("The following files are in list1 but not list2:");
            foreach (var v in queryList1Only)
            {
                Console.WriteLine(v.FullName);
            }

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    // This implementation defines a very simple comparison
    // between two FileInfo objects. It only compares the name
    // of the files being compared and their length in bytes.
    class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
    {
        public FileCompare() { }

        public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
        {
            return (f1.Name == f2.Name &&
                    f1.Length == f2.Length);
        }

        // Return a hash that reflects the comparison criteria. According to the
        // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
        // also be equal. Because equality as defined here is a simple value equality, not
        // reference identity, it is possible that two or more objects will produce the same
        // hash code.
        public int GetHashCode(System.IO.FileInfo fi)
        {
            string s = $"{fi.Name}{fi.Length}";
            return s.GetHashCode();
        }
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Comment interroger les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)

02/11/2020 • 5 minutes to read • Edit Online

Cet exemple montre cinq requêtes liées à la taille des fichiers en octets :

- Comment récupérer la taille en octets du plus grand fichier.
- Comment récupérer la taille en octets du plus petit fichier.
- Comment récupérer le plus grand ou le plus petit fichier de l'objet FileInfo dans un ou plusieurs dossiers situés dans le dossier racine spécifié.
- Comment récupérer une séquence, telle que les 10 fichiers les plus volumineux.
- Comment regrouper des fichiers selon leur taille en octets, en ignorant les fichiers qui sont inférieurs à une taille spécifiée.

Exemple

L'exemple suivant contient cinq requêtes distinctes qui montrent comment interroger et regrouper des fichiers selon leur taille en octets. Vous pouvez facilement modifier ces exemples pour baser la requête sur une autre propriété de l'objet FileInfo.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
             .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
            (from file in fileList
```

```
            let len = GetFileLength(file)
            where len > 0
            orderby len descending
            select file)
           .First();

        Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                          startFolder, longestFile.FullName, longestFile.Length);

        //Return the FileInfo of the smallest file
        System.IO.FileInfo smallestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len ascending
             select file).First();

        Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                          startFolder, smallestFile.FullName, smallestFile.Length);

        //Return the FileInfos for the 10 largest files
        // queryTenLargest is an IEnumerable<System.IO.FileInfo>
        var queryTenLargest =
            (from file in fileList
             let len = GetFileLength(file)
             orderby len descending
             select file).Take(10);

        Console.WriteLine("The 10 largest files under {0} are:", startFolder);

        foreach (var v in queryTenLargest)
        {
            Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
        }

        // Group the files according to their size, leaving out
        // files that are less than 200000 bytes.
        var querySizeGroups =
            from file in fileList
            let len = GetFileLength(file)
            where len > 0
            group file by (len / 100000) into fileGroup
            where fileGroup.Key >= 2
            orderby fileGroup.Key descending
            select fileGroup;

        foreach (var filegroup in querySizeGroups)
        {
            Console.WriteLine(filegroup.Key.ToString() + "00000");
            foreach (var item in filegroup)
            {
                Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
            }
        }
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the FileInfo.Length property.
    // In this particular case, it is safe to swallow the exception.
    static long GetFileLength(System.IO.FileInfo fi)
    {
        long retval;
        try
        {
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
```

```
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }

}
```

Pour retourner un ou plusieurs objets FileInfo complets, la requête doit d'abord examiner chacun d'eux dans la source de données, puis les trier selon la valeur de leur propriété Length. Elle peut ensuite retourner l'objet ou la séquence d'objets avec les plus grandes valeurs de longueur. Utilisez First pour retourner le premier élément d'une liste. Utilisez Take pour retourner les n premiers éléments. Spécifiez un ordre de tri décroissant pour placer les plus petits éléments au début de la liste.

La requête appelle une méthode distincte pour obtenir la taille du fichier en octets et ainsi permettre l'utilisation de l'exception éventuellement levée si un fichier a été supprimé sur un autre thread depuis la création de l'objet FileInfo dans l'appel à `GetFiles`. Même si l'objet FileInfo a déjà été créé, l'exception peut être levée, car un objet FileInfo essaiera d'actualiser sa propriété Length en utilisant la taille en octets la plus récente lors du premier accès à la propriété. En plaçant cette opération dans un bloc try-catch en dehors de la requête, nous respectons la règle qui consiste à éviter les opérations dans les requêtes qui peuvent avoir des effets secondaires. En règle générale, il faut faire très attention lors de l'utilisation d'exceptions et s'assurer que l'application ne reste pas dans un état inconnu.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Comment interroger des fichiers dupliqués dans une arborescence de répertoires (LINQ) (C#)

02/11/2020 • 4 minutes to read • Edit Online

Parfois, plusieurs dossiers peuvent contenir des fichiers ayant le même nom. Par exemple, sous le dossier d'installation de Visual Studio, plusieurs dossiers ont un fichier readme.htm. Cet exemple montre comment rechercher ces noms de fichiers dupliqués sous un dossier racine spécifié. Le deuxième exemple montre comment interroger les fichiers dont la taille et les durées de dernière écriture correspondent également.

Exemple

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
        public long Length { get; set; }
```

```
        public override bool Equals(object obj)
        {
            PortableKey other = (PortableKey)obj;
            return other.LastWriteTime == this.LastWriteTime &&
                   other.Length == this.Length &&
                   other.Name == this.Name;
        }

        public override int GetHashCode()
        {
            string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
            return str.GetHashCode();
        }
        public override string ToString()
        {
            return $"{this.Name} {this.Length} {this.LastWriteTime}";
        }
    }
    static void QueryDuplicates2()
    {
        // Change the root drive or folder if necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Make the lines shorter for the console display
        int charsToSkip = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // Note the use of a compound key. Files that match
        // all three properties belong to the same group.
        // A named type is used to enable the query to be
        // passed to another method. Anonymous types can also be used
        // for composite keys but cannot be passed across method boundaries
        //
        var queryDupFiles =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by
                new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length }
into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        var list = queryDupFiles.ToList();

        int i = queryDupFiles.Count();

        PageOutput<PortableKey, string>(queryDupFiles);
    }

    // A generic method to page the output of the QueryDuplications methods
    // Here the type of the group must be specified explicitly. "var" cannot
    // be used in method signatures. This method does not display more than one
    // group per page.
    private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;
```

```
            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var fileName in resultPage)
                {
                    Console.WriteLine("\t{0}", fileName);
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

La première requête utilise une clé simple pour déterminer une correspondance ; elle trouve les fichiers qui ont le même nom mais dont le contenu peut être différent. La deuxième requête utilise une clé composée à comparer à trois propriétés de l'objet FileInfo. Cette requête est beaucoup plus susceptible de trouver les fichiers ayant le même nom et un contenu similaire ou identique.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ to Objects (C#)
- LINQ et répertoires de fichiers (C#)

Comment interroger le contenu de fichiers texte dans un dossier (LINQ) (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment interroger tous les fichiers d'une arborescence de répertoires spécifiée, comment ouvrir chaque fichier et comment inspecter son contenu. Ce type de technique peut être utilisé pour créer des index ou des index inversés à partir du contenu d'une arborescence de répertoires. Dans cet exemple, une recherche de chaîne simple est effectuée. Toutefois, il est possible d'effectuer une recherche avec des critères spéciaux plus complexes à l'aide d'une expression régulière. Pour plus d'informations, consultez Guide pratique pour combiner des requêtes LINQ avec des expressions régulières (C#).

Exemple

```
class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- LINQ et répertoires de fichiers (C#)
- LINQ to Objects (C#)

Comment interroger un ArrayList avec LINQ (C#)

02/11/2020 • 2 minutes to read • Edit Online

Quand vous utilisez LINQ pour interroger des collections IEnumerable non génériques telles que ArrayList, vous devez déclarer explicitement le type de la variable de portée pour qu'il reflète le type spécifique des objets de la collection. Par exemple, si vous avez un ArrayList d'objets `Student`, votre clause from doit ressembler à ceci :

```
var query = from Student s in arrList
//...
```

En spécifiant le type de la variable de portée, vous effectuez un cast de chaque élément du `Student` en ArrayList.

L'utilisation d'une variable de portée explicitement typée dans une expression de requête équivaut à appeler la méthode Cast. Cast lève une exception si le cast spécifié ne peut pas être effectué. Cast et OfType sont les deux méthodes d'opérateur de requête standard qui fonctionnent sur les types IEnumerable non génériques. Pour plus d'informations, consultez relations de types dans les opérations de requête LINQ.

Exemple

L'exemple suivant montre une requête simple sur un ArrayList. Notez que cet exemple utilise des initialiseurs d'objets quand le code appelle la méthode Add, mais cela n'est pas une obligation.

```
using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                    {
                        FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60
}
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                    });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
/* Output:
    Omelchenko: 98
    Garcia: 97
*/
```

Voir aussi

- LINQ to Objects (C#)

Comment ajouter des méthodes personnalisées pour les requêtes LINQ (C#)

02/11/2020 • 8 minutes to read • Edit Online

Vous étendez l'ensemble de méthodes que vous utilisez pour les requêtes LINQ en ajoutant des méthodes d'extension à l' IEnumerable<T> interface. Par exemple, en plus des opérations standard de moyenne ou maximale, vous créez une méthode d'agrégation personnalisée pour calculer une valeur unique à partir d'une séquence de valeurs. Vous créez également une méthode qui fonctionne comme un filtre personnalisé ou une transformation de données spécifique pour une séquence de valeurs et retourne une nouvelle séquence. Distinct, Skip et Reverse en sont quelques exemples.

Quand vous étendez l'interface IEnumerable<T>, vous pouvez appliquer vos méthodes personnalisées à n'importe quelle collection énumérable. Pour plus d'informations, consultez Méthodes d'extension.

Ajout d'une méthode d'agrégation

Une méthode d'agrégation calcule une valeur à partir d'un ensemble de valeurs. LINQ fournit plusieurs méthodes d'agrégation, notamment Average, Min et Max. Vous pouvez créer votre propre méthode d'agrégation en ajoutant une méthode d'extension à l'interface IEnumerable<T>.

L'exemple de code suivant montre comment créer une méthode d'extension appelée `Median` pour calculer une valeur médiane pour une séquence de nombres de type `double`.

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (!(source?.Any() ?? false))
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList = (from number in source
                          orderby number
                          select number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Vous appelez cette méthode d'extension pour toute collection énumérable de la même façon que vous appelez d'autres méthodes d'agrégation depuis l'interface IEnumerable<T>.

L'exemple de code suivant montre comment utiliser la méthode `Median` pour un tableau de type `double`.

```
double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query = numbers.Median();

Console.WriteLine("double: Median = " + query);
/*
 This code produces the following output:

 Double: Median = 4.85
*/
```

Surcharge d'une méthode d'agrégation pour accepter différents types

Vous pouvez surcharger votre méthode d'agrégation pour qu'elle accepte des séquences de différents types. L'approche standard consiste à créer une surcharge pour chaque type. Une autre approche consiste à créer une surcharge qui accepte un type générique et le convertit en un autre type à l'aide d'un délégué. Vous pouvez également combiner ces deux approches.

Pour créer une surcharge pour chaque type

Vous pouvez créer une surcharge pour chacun des types que vous voulez prendre en charge. L'exemple de code suivant montre une surcharge de la méthode `Median` pour le type `int`.

```
//int overload
public static double Median(this IEnumerable<int> source) =>
    (from num in source select (double)num).Median();
```

Vous pouvez maintenant appeler les surcharges `Median` pour les types `integer` et `double`, comme indiqué dans le code suivant :

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);

int[] numbers2 = { 1, 2, 3, 4, 5 };

var query2 = numbers2.Median();

Console.WriteLine("int: Median = " + query2);
/*
 This code produces the following output:

 Double: Median = 4.85
 Integer: Median = 3
*/
```

Pour créer une surcharge générique

Vous pouvez également créer une surcharge qui accepte une séquence d'objets génériques. Cette surcharge prend un délégué comme paramètre et l'utilise pour convertir une séquence d'objets de type générique en un autre type d'objets.

Le code suivant montre une surcharge de la méthode `Median` qui prend le délégué Func<T,TResult> comme paramètre. Ce délégué prend un objet de type générique T et retourne un objet de type `double`.

```
// Generic overload.
public static double Median<T>(this IEnumerable<T> numbers,
                        Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Vous pouvez maintenant appeler la méthode `Median` pour une séquence d'objets de tout type. Si le type n'a pas sa propre surcharge de méthode, vous devez passer un paramètre de délégué. En C#, vous pouvez utiliser une expression lambda à cet effet. En outre, en Visual Basic uniquement, si vous utilisez la clause `Aggregate` ou `Group By` au lieu de l'appel de méthode, vous pouvez passer n'importe quelle valeur ou expression qui se trouve dans la portée de cette clause.

L'exemple de code suivant montre comment appeler la méthode `Median` pour un tableau d'entiers et un tableau de chaînes. Pour les chaînes, c'est la valeur médiane des longueurs de chaînes du tableau qui est calculée. L'exemple montre comment passer le paramètre de délégué Func<T,TResult> à la méthode `Median` pour chaque cas.

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
  You can use the num=>num lambda expression as a parameter for the Median method
  so that the compiler will implicitly convert its value to double.
  If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine("int: Median = " + query3);

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.

var query4 = numbers4.Median(str => str.Length);

Console.WriteLine("String: Median = " + query4);

/*
 This code produces the following output:

 Integer: Median = 3
 String: Median = 4
*/
```

Ajout d'une méthode qui retourne une séquence

Vous pouvez étendre l'interface IEnumerable<T> avec une méthode de requête personnalisée qui retourne une séquence de valeurs. Dans ce cas, la méthode doit retourner une collection de type IEnumerable<T>. Ces méthodes peuvent être utilisées pour appliquer des filtres ou des transformations de données à une séquence de valeurs.

L'exemple suivant montre comment créer une méthode d'extension nommée `AlternateElements` qui retourne un élément sur deux d'une collection, en commençant par le premier élément.

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int i = 0;
    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            yield return element;
        }
        i++;
    }
}
```

Vous pouvez appeler cette méthode d'extension pour n'importe quelle collection énumérable, de la même façon que vous appelez d'autres méthodes depuis l'interface IEnumerable<T>, comme le montre le code suivant :

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
/*
 This code produces the following output:

 a
 c
 e
*/
```

Voir aussi

- IEnumerable<T>
- Méthodes d'extension

LINQ to ADO.NET (page de portail)

02/11/2020 • 3 minutes to read • Edit Online

LINQ to ADO.NET vous permet d'interroger un objet énumérable dans ADO.NET à l'aide du modèle de programmation LINQ (Language-Integrated Query).

> **NOTE**
> La documentation LINQ to ADO.NET se trouve dans la section ADO.NET du kit de développement logiciel (SDK) .NET Framework : LINQ et ADO.net.

Il existe trois technologies ADO.NET LINQ (Language-Integrated Query) distinctes : LINQ to DataSet, LINQ to SQL et LINQ to Entities. LINQ to DataSet assure une interrogation plus riche et optimisée du DataSet, LINQ to SQL vous permet d'interroger directement des schémas de base de données SQL Server, et LINQ to Entities vous permet d'interroger un Entity Data Model.

LINQ to DataSet

DataSet est l'un des composants les plus largement utilisés dans ADO.NET, et c'est un élément clé du modèle de programmation déconnecté sur lequel ADO.NET est fondé. Toutefois, en dépit de son importance, le DataSet a des capacités de requête limitées.

LINQ to DataSet vous permet de générer des fonctionnalités de requête plus complètes dans DataSet en utilisant les mêmes fonctionnalités de requête qui sont disponibles pour de nombreuses autres sources de données.

Pour plus d'informations, consultez LINQ to DataSet.

LINQ to SQL

LINQ to SQL fournit une infrastructure runtime pour la gestion des données relationnelles comme objets. Dans LINQ to SQL, le modèle de données d'une base de données relationnelle est mappé à un modèle objet exprimé dans le langage de programmation du développeur. Lors de l'exécution de l'application, LINQ to SQL traduit les requêtes intégrées au langage du modèle objet en SQL et les envoie à la base de données pour exécution. Quand la base de données retourne les résultats, LINQ to SQL les retraduit en objets que vous pouvez utiliser.

LINQ to SQL inclut la prise en charge des procédures stockées et fonctions définies par l'utilisateur dans la base de données, et de l'héritage dans le modèle objet.

Pour plus d'informations, consultez LINQ to SQL.

LINQ to Entities

Via l'Entity Data Model, les données relationnelles sont exposées comme objets dans l'environnement .NET. Cela fait de la couche objet une cible idéale pour la prise en charge de LINQ, permettant aux développeurs de formuler des requêtes sur la base de données à partir du langage utilisé pour générer la logique métier. Cette capacité est appelée LINQ to Entities. Pour plus d'informations, consultez LINQ to Entities.

Voir aussi

- LINQ et ADO.NET
- LINQ (Language-Integrated Query) (C#)

Activation d'une source de données pour l'interrogation LINQ

02/11/2020 • 6 minutes to read • Edit Online

Il existe plusieurs façons d'étendre LINQ pour permettre l'interrogation de toute source de données dans le modèle LINQ. La source de données peut être une structure de données, un service Web, un système de fichiers ou une base de données, par exemple. Le modèle LINQ permet aux clients d'interroger facilement une source de données pour laquelle l'interrogation LINQ est activée, car la syntaxe et le modèle de la requête ne changent pas. Les façons dont LINQ peut être étendu à ces sources de données sont les suivantes :

- Implémentation de l' IEnumerable<T> interface dans un type pour permettre LINQ to Objects interrogation de ce type.
- Création de méthodes d'opérateur de requête standard telles que Where et Select qui étendent un type, pour activer l'interrogation LINQ personnalisée de ce type.
- Création d'un fournisseur pour votre source de données qui implémente l'interface IQueryable<T>. Un fournisseur qui implémente cette interface reçoit des requêtes LINQ sous la forme d'arborescences d'expression, qu'il peut exécuter de façon personnalisée, par exemple à distance.
- Création d'un fournisseur pour votre source de données qui tire parti d'une technologie LINQ existante. Ce type de fournisseur permettrait non seulement l'interrogation, mais également l'insertion, la mise à jour et la suppression d'opérations et le mappage pour des types définis par l'utilisateur.

Cette rubrique décrit ces options.

Comment activer l'interrogation LINQ de votre source de données

Données en mémoire

Vous pouvez activer l'interrogation LINQ des données en mémoire de deux manières. Si les données sont d'un type qui implémente IEnumerable<T> , vous pouvez interroger les données à l'aide de LINQ to Objects. S'il est inutile d'activer l'énumération de votre type en implémentant l' IEnumerable<T> interface, vous pouvez définir des méthodes d'opérateur de requête standard LINQ dans ce type ou créer des méthodes d'opérateur de requête standard LINQ qui étendent le type. Les implémentations personnalisées des opérateurs de requête standard doivent utiliser l'exécution différée pour retourner les résultats.

Données distantes

La meilleure option pour activer l'interrogation LINQ d'une source de données distante est d'implémenter l' IQueryable<T> interface. Toutefois, cette méthode diffère de l'extension d'un fournisseur tel que LINQ to SQL pour une source de données. Aucun modèle de fournisseur pour l'extension des technologies LINQ existantes, tel que LINQ to SQL , à d'autres types de source de données n'est disponible dans Visual Studio 2008.

Fournisseurs LINQ IQueryable

Les fournisseurs LINQ qui implémentent IQueryable<T> peuvent varier considérablement dans leur complexité. Cette section décrit les différents niveaux de complexité.

Un fournisseur `IQueryable` moins complexe peut interagir avec une méthode unique d'un service Web. Ce type de fournisseur est très particulier, car il attend des informations spécifiques dans les requêtes qu'il gère. Il se caractérise par un système de type fermé, exposant peut-être un type de résultat unique. La plus grande part de

l'exécution de la requête est effectuée localement, par exemple à l'aide des implémentations Enumerable des opérateurs de requête standard. Un fournisseur moins complexe peut examiner une seule expression d'appel de méthode dans l'arborescence d'expression qui représente la requête, et la logique restante de la requête doit être gérée ailleurs.

Un fournisseur `IQueryable` de complexité moyenne peut cibler une source de données qui possède un langage de requête partiellement expressif. S'il cible un service Web, il peut interagir avec plusieurs méthodes du service Web et sélectionner la méthode d'appel en fonction de la question posée par la requête. Un fournisseur de complexité moyenne aurait un système plus sophistiqué que celui d'un fournisseur simple, qui serait néanmoins un système de type fixe. Par exemple, le fournisseur peut exposer des types qui ont des relations un-à-plusieurs qui peuvent être parcourues, mais il ne fournirait pas de technologie de mappage pour les types définis par l'utilisateur.

Un `IQueryable` fournisseur complexe, tel que le LINQ to SQL fournisseur, peut traduire des requêtes LINQ complètes en un langage de requête expressif, tel que SQL. Un fournisseur complexe est plus général qu'un fournisseur moins complexe, car il peut gérer une plus grande gamme de questions dans la requête. Il possède également un système de type ouvert et doit donc contenir une infrastructure étendue pour mapper les types définis par l'utilisateur. Développer un fournisseur complexe est une tâche difficile.

Voir aussi

- IQueryable<T>
- IEnumerable<T>
- Enumerable
- Présentation des opérateurs de requête standard (C#)
- LINQ to Objects (C#)

Prise en charge de LINQ par les outils et l'IDE Visual Studio (C#)

02/11/2020 • 2 minutes to read • Edit Online

L'environnement de développement intégré (IDE) Visual Studio fournit les fonctionnalités suivantes qui prennent en charge le développement d'applications LINQ :

Object Relational Designer

L'outil Concepteur Objet Relationnel est un outil de conception visuelle que vous pouvez utiliser dans les applications LINQ to SQL pour générer des classes en C# qui représentent les données relationnelles d'une base de données sous-jacente. Pour plus d'informations, consultez Outils LINQ to SQL dans Visual Studio.

Outil en ligne de commande SQLMetal

SQLMetal est un outil en ligne de commande qui peut être utilisé dans les processus de génération pour créer des classes à partir de bases de données existantes pour une utilisation dans les applications LINQ to SQL. Pour plus d'informations, consultez SqlMetal.exe (outil de génération de code).

Éditeur de code prenant en charge LINQ

L'éditeur de code C# prend largement en charge LINQ avec IntelliSense et des fonctionnalités de mise en forme.

Prise en charge du débogueur Visual Studio

Le débogueur Visual Studio prend en charge le débogage d'expressions de requête. Pour plus d'informations, consultez Débogage LINQ.

Voir aussi

- LINQ (Language-Integrated Query) (C#)

Réflexion (C#)

02/11/2020 • 3 minutes to read • Edit Online

La réflexion fournit des objets (de type Type) qui décrivent les assemblys, les modules et les types. Vous pouvez utiliser la réflexion pour créer dynamiquement une instance d'un type, lier le type à un objet existant ou obtenir le type à partir d'un objet existant et invoquer ses méthodes ou accéder à ses champs et propriétés. Si vous utilisez des attributs dans votre code, la réflexion vous permet d'y accéder. Pour plus d'informations, consultez Attributs.

Voici un exemple simple de réflexion à l'aide de la GetType() méthode, héritée par tous les types de la `Object` classe de base, pour obtenir le type d'une variable :

> **NOTE**
>
> Veillez à ajouter `using System;` et `using System.Reflection;` en haut de votre fichier *.cs* .

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

La sortie est : `System.Int32` .

L'exemple suivant utilise la réflexion pour obtenir le nom complet de l'assembly chargé.

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

La sortie est : `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e` .

> **NOTE**
>
> Les mots clés C# `protected` et `internal` n'ont aucune signification en langage intermédiaire et ne sont pas utilisés dans les API de réflexion. Les termes correspondants en langage intermédiaire sont *Family* et *Assembly*. Pour identifier une méthode `internal` à l'aide de la réflexion, utilisez la propriété IsAssembly. Pour identifier une méthode `protected internal`, utilisez IsFamilyOrAssembly.

Vue d'ensemble de la réflexion

La réflexion est utile dans les situations suivantes :

- Lorsque vous devez accéder à des attributs dans les métadonnées de votre programme. Pour plus d'informations, consultez Récupération des informations stockées dans les attributs.
- Pour examiner et instancier des types dans un assembly.
- Pour créer de nouveaux types au moment de l'exécution. Utilisez des classes dans System.Reflection.Emit.
- Pour effectuer une liaison tardive, en accédant aux méthodes sur les types créés au moment de l'exécution. Consultez la rubrique Chargement et utilisation dynamiques des types.

Sections connexes

Pour plus d'informations :

- Réflexion
- Affichage des informations de type
- Réflexion et types génériques
- System.Reflection.Emit
- Récupération des informations stockées dans les attributs

Voir aussi

- Guide de programmation C#
- Assemblys dans .NET

Sérialisation (C#)

02/11/2020 • 9 minutes to read • Edit Online

La sérialisation est le processus de conversion d'un objet en flux d'octets pour stocker l'objet ou le transmettre à la mémoire, une base de données ou un fichier. Son principal objectif est d'enregistrer l'état d'un objet afin de pouvoir le recréer si nécessaire. Le processus inverse est appelé désérialisation.

Fonctionnement de la sérialisation

Cette illustration montre le processus global de sérialisation :

L'objet est sérialisé en un flux qui contient les données. Le flux peut également contenir des informations sur le type de l'objet, telles que sa version, sa culture et son nom d'assembly. À partir de ce flux, l'objet peut être stocké dans une base de données, un fichier ou une mémoire.

Usages de la sérialisation

La sérialisation permet au développeur d'enregistrer l'état d'un objet et de le recréer en fonction des besoins, en fournissant le stockage des objets ainsi que l'échange de données. Par le biais de la sérialisation, un développeur peut effectuer des actions telles que :

- Envoi de l'objet à une application distante à l'aide d'un service Web
- Passage d'un objet d'un domaine à un autre
- Passage d'un objet via un pare-feu sous forme de chaîne JSON ou XML
- Gestion des informations relatives à la sécurité ou aux utilisateurs entre les applications

Sérialisation JSON

L' System.Text.Json espace de noms contient des classes pour la sérialisation et la désérialisation JavaScript Object Notation (JSON). JSON est une norme ouverte qui est couramment utilisée pour partager des données sur le Web.

La sérialisation JSON sérialise les propriétés publiques d'un objet dans une chaîne, un tableau d'octets ou un flux conforme à la spécification JSON RFC 8259. Pour contrôler la manière dont JsonSerializer sérialise ou désérialise une instance de la classe :

- Utiliser un JsonSerializerOptions objet
- Appliquer des attributs de l' System.Text.Json.Serialization espace de noms à des classes ou des propriétés
- Implémenter des convertisseurs personnalisés

Sérialisation binaire et XML

L' System.Runtime.Serialization espace de noms contient des classes pour la sérialisation et la désérialisation binaires et XML.

La sérialisation binaire utilise l'encodage binaire pour produire une sérialisation compacte qui peut servir notamment aux flux réseau par socket ou stockage. Dans la sérialisation binaire, tous les membres, y compris en lecture seule, sont sérialisés et les performances sont améliorées.

> **WARNING**
>
> La sérialisation binaire peut être dangereuse. Pour plus d'informations, consultez le Guide de la sécurité BinaryFormatter.

La sérialisation XML sérialise les champs et les propriétés publics d'un objet ou les paramètres et valeurs renvoyés de méthodes, en un flux XML conforme à un document XSD (langage de définition de schéma XML) spécifique. La sérialisation XML permet d'obtenir des classes fortement typées avec des propriétés et des champs publics convertis au format XML. System.Xml.Serializationcontient des classes pour sérialiser et désérialiser du code XML. Vous pouvez appliquer des attributs à des classes et des membres de classes pour contrôler la manière dont XmlSerializer sérialise ou désérialise une instance de la classe.

Rendre un objet sérialisable

Pour la sérialisation binaire ou XML, vous avez besoin des éléments suivants :

- Objet à sérialiser.
- Flux destiné à contenir l'objet sérialisé
- Une System.Runtime.Serialization.Formatter instance

Appliquez l' SerializableAttribute attribut à un type pour indiquer que les instances du type peuvent être sérialisées. Une exception est levée si vous tentez une sérialisation alors que le type n'a pas l'attribut SerializableAttribute.

Pour empêcher la sérialisation d'un champ, appliquez l' NonSerializedAttribute attribut. Si un champ d'un type sérialisable contient un pointeur, un handle ou une autre structure de données propre à un environnement particulier et qu'il n'est pas possible de le reconstituer de manière significative dans un autre environnement, vous pouvez le rendre non sérialisable.

Si une classe sérialisée contient des références à des objets d'autres classes marqués par SerializableAttribute, ces objets seront également sérialisés.

Sérialisation de base et sérialisation personnalisée

La sérialisation binaire et XML peut être effectuée de deux manières, de base et personnalisée.

La sérialisation de base utilise .NET pour sérialiser automatiquement l'objet. La seule exigence est que l'attribut est appliqué à la classe SerializableAttribute . NonSerializedAttribute peut être utilisé pour empêcher que certains champs ne soient sérialisés.

Quand vous utilisez la sérialisation de base, le contrôle de version des objets peut créer des problèmes. Vous pouvez utiliser la sérialisation personnalisée quand les problèmes de contrôle de version sont importants. La sérialisation de base est le moyen le plus simple d'effectuer la sérialisation, mais elle ne fournit pas beaucoup de contrôle sur le processus.

Dans la sérialisation personnalisée, vous pouvez spécifier exactement quels objets sont sérialisés et de quelle façon. La classe doit être marquée SerializableAttribute et implémenter l'interface ISerializable. Si vous souhaitez également que votre objet soit désérialisé de manière personnalisée, utilisez un constructeur personnalisé.

Sérialisation du concepteur

La sérialisation du concepteur est une forme particulière de sérialisation qui implique le type de persistance d'objets associé aux outils de développement. La sérialisation du concepteur est le processus qui consiste à convertir un graphique d'objet en un fichier source utilisable ultérieurement pour récupérer le graphique d'objet.

Un fichier source peut contenir du code, des balises ou même des tables SQL.

Rubriques connexes et exemples

Présentation deSystem.Text.Js Montre comment récupérer la `System.Text.Json` bibliothèque.

Comment sérialiser et désérialiser JSON dans .net. Montre comment lire et écrire des données d'objet dans et à partir de JSON à l'aide de la JsonSerializer classe.

Procédure pas à pas : persistance d'un objet dans Visual Studio (C#)
Montre comment utiliser la sérialisation pour rendre les données d'un objet persistantes entre les instances, afin de stocker des valeurs et de les récupérer lors de la prochaine instanciation de l'objet.

Comment lire des données d'objet à partir d'un fichier XML (C#)
Montre comment lire les données d'objet écrites précédemment dans un fichier XML en utilisant la classe XmlSerializer.

Comment écrire des données d'objet dans un fichier XML (C#)
Montre comment écrire l'objet depuis une classe vers un fichier XML en utilisant la classe XmlSerializer.

Comment écrire des données d'objet dans un fichier XML (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple écrit l'objet d'une classe dans un fichier XML en utilisant la classe XmlSerializer.

Exemple

```
public class XMLWrite
{

   static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
"//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compilation du code

La classe en cours de sérialisation doit disposer d'un constructeur public sans paramètres.

Programmation fiable

Les conditions ci-dessous peuvent générer une exception.

- La classe qui est sérialisée n'a pas de constructeur public sans paramètres.
- Le fichier existe et est en lecture seule (IOException).
- Le chemin est trop long (PathTooLongException).
- Le disque est plein (IOException).

Sécurité .NET

Cet exemple crée un fichier s'il n'existe pas déjà. Si une application doit créer un fichier, elle doit disposer de l'autorisation `Create` pour accéder au dossier. Si le fichier existe déjà, l'application a uniquement besoin de l'autorisation `Write`, qui est une autorisation de niveau inférieur. Quand cela est possible, il est plus sûr de créer le fichier au cours du déploiement et de n'accorder l'autorisation `Read` que sur un seul fichier, plutôt que l'autorisation `Create` sur un dossier.

Voir aussi

- StreamWriter
- Comment lire des données d'objet à partir d'un fichier XML (C#)
- Sérialisation (C#)

Comment lire des données d'objet à partir d'un fichier XML (C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple lit des données d'objet écrites précédemment dans un fichier XML en utilisant la classe XmlSerializer.

Exemple

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview =  (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);

}
```

Compilation du code

Remplacez le nom du fichier « c:\temp\SerializationOverview.xml » par le nom du fichier qui contient les données sérialisées. Pour plus d'informations sur la sérialisation des données, consultez Comment écrire des données d'objet dans un fichier XML (C#).

La classe doit disposer d'un constructeur public sans paramètres.

Seuls les propriétés et les champs publics sont désérialisés.

Programmation fiable

Les conditions ci-dessous peuvent générer une exception.

- La classe qui est sérialisée n'a pas de constructeur public sans paramètres.
- Les données du fichier ne représentent pas les données de la classe à désérialiser.
- Le fichier n'existe pas (IOException).

Sécurité .NET

Vérifiez toujours les entrées et ne désérialisez jamais les données provenant d'une source non fiable. L'objet recréé s'exécute sur un ordinateur local avec les autorisations du code qui l'a désérialisé. Vérifiez toutes les entrées avant d'utiliser les données dans votre application.

Voir aussi

- StreamWriter
- Comment écrire des données d'objet dans un fichier XML (C#)
- Sérialisation (C#)
- Guide de programmation C#

Procédure pas à pas : Rendre un objet persistant avec C#

02/11/2020 • 9 minutes to read • Edit Online

Vous pouvez utiliser la sérialisation pour rendre les données d'un objet persistantes entre les instances, ce qui vous permet de stocker des valeurs et de les récupérer lors de la prochaine instanciation de l'objet.

Dans cette procédure pas à pas, vous créez un objet `Loan` de base et vous stockez ses données dans un fichier. Vous récupérerez ensuite les données du fichier lors de la recréation de l'objet.

> **IMPORTANT**
>
> Cet exemple crée un fichier s'il n'existe pas déjà. Si une application doit créer un fichier, elle doit disposer de l'autorisation `Create` pour le dossier. Les autorisations sont définies à l'aide de listes de contrôle d'accès. Si le fichier existe déjà, l'application a uniquement besoin de l'autorisation `Write`, ce qui représente une autorisation inférieure. Quand c'est possible, il est plus sûr de créer le fichier lors du déploiement et d'accorder l'autorisation `Read` sur un seul fichier (au lieu d'accorder l'autorisation Créer pour un dossier). Par ailleurs, il est plus sûr d'écrire les données dans des dossiers utilisateur que dans le dossier racine ou le dossier Program Files.

> **IMPORTANT**
>
> Cet exemple stocke les données dans un fichier au format binaire. Ces formats ne doivent pas être utilisés pour des données sensibles, telles que les mots de passe ou les informations relatives à la carte de crédit.

Prérequis

- Pour générer et exécuter, installez le SDK .NET Core.
- Installez votre éditeur de code favori, si ce n'est pas déjà fait.

> **TIP**
>
> Vous avez besoin d'installer un éditeur de code ? Essayez Visual Studio !

- L'exemple exige C# 7.3. Voir Sélectionner la version du langage C#.

Vous pouvez examiner l'exemple de code en ligne sur le référentiel GitHub des exemples .NET.

Création de l'objet loan

La première étape consiste à créer une classe `Loan` et une application console qui utilise cette classe :

1. Créez une application. Tapez `dotnet new console -o serialization` pour créer une application console dans un sous-répertoire nommé `serialization`.
2. Ouvrez l'application dans votre éditeur, et ajoutez une nouvelle classe nommée `Loan.cs`.
3. Ajoutez le code suivant à la classe `Loan` :

```
public class Loan : INotifyPropertyChanged
 {
     public double LoanAmount { get; set; }
     public double InterestRatePercent { get; set; }

     [field:NonSerialized()]
     public DateTime TimeLastLoaded { get; set; }

     public int Term { get; set; }

     private string customer;
     public string Customer
     {
         get { return customer; }
         set
         {
             customer = value;
             PropertyChanged?.Invoke(this,
               new PropertyChangedEventArgs(nameof(Customer)));
         }
     }

     [field: NonSerialized()]
     public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

     public Loan(double loanAmount,
                 double interestRate,
                 int term,
                 string customer)
     {
         this.LoanAmount = loanAmount;
         this.InterestRatePercent = interestRate;
         this.Term = term;
         this.customer = customer;
     }
 }
```

Vous devez également créer une application simple qui utilise la classe `Loan`.

Sérialiser l'objet loan

1. Ouvrez `Program.cs`. Ajoutez le code suivant :

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

Ajoutez un gestionnaire d'événements pour l'événement `PropertyChanged`, ainsi que quelques lignes pour modifier l'objet `Loan` et afficher les modifications. Vous pouvez voir les ajouts dans le code suivant :

```
TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);
```

À ce stade, vous pouvez exécuter le code et voir la sortie actuelle :

```
New customer value: Henry Clay
7.5
7.1
```

L'exécution de cette application à plusieurs reprises écrit toujours les mêmes valeurs. Un nouvel objet Loan est créé chaque fois que vous exécutez le programme. Dans la réalité, les taux d'intérêt changent régulièrement, mais pas nécessairement chaque fois que l'application est exécutée. Un code de sérialisation signifie que vous conservez le taux d'intérêt le plus récent entre les instances de l'application. C'est ce que vous ferez à l'étape suivante, en ajoutant la sérialisation à la classe Loan.

Utilisation de la sérialisation pour rendre l'objet persistant

Pour rendre les valeurs de la classe Loan persistantes, vous devez d'abord marquer la classe avec l'attribut `Serializable`. Ajoutez le code suivant au-dessus de la déclaration de la classe Loan :

```
[Serializable()]
```

SerializableAttribute indique au compilateur que tout ce que contient la classe peut être stocké dans un fichier. Étant donné que l'événement `PropertyChanged` ne représente pas une partie du graphe d'objets qui doit être stocké, il ne doit pas être sérialisé. Procéder ainsi sérialiserait tous les objets qui sont attachés à cet événement. Vous pouvez ajouter NonSerializedAttribute à la déclaration du champ pour le gestionnaire d'événements `PropertyChanged`.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

À compter de C# 7.3, vous pouvez attacher des attributs au champ de stockage d'une propriété implémentée automatiquement en utilisant la valeur cible `field`. Le code suivant ajoute une propriété `TimeLastLoaded` et la marque comme non sérialisable :

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

L'étape suivante consiste à ajouter le code de sérialisation à l'application LoanApp. Pour sérialiser la classe et l'écrire dans un fichier, vous utilisez les espaces de noms System.IO et System.Runtime.Serialization.Formatters.Binary. Pour éviter de taper les noms qualifiés complets, vous pouvez ajouter des références aux espaces de noms nécessaires, comme illustré dans le code suivant :

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

L'étape suivante consiste à ajouter le code permettant de désérialiser l'objet à partir du fichier lors de la création de l'objet. Ajoutez une constante à la classe pour le nom de fichier des données sérialisées, comme illustré dans le code suivant :

```
const string FileName = @"../../../SavedLoan.bin";
```

Ensuite, ajoutez le code suivant après la ligne qui crée l'objet `TestLoan` :

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

Vous devez d'abord vérifier que le fichier existe. S'il existe, créez une classe Stream pour lire le fichier binaire et une classe BinaryFormatter pour convertir le fichier. Vous devez également convertir le type de flux en type d'objet Loan.

Vous devez ensuite ajouter du code pour sérialiser la classe dans un fichier. Ajoutez le code suivant après le code existant de la méthode `Main` :

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

À ce stade, vous pouvez de nouveau générer et exécuter l'application. La première fois qu'elle s'exécute, notez que le taux d'intérêt commence à 7,5, puis qu'il passe à 7,1. Fermez l'application et exécutez-la de nouveau. L'application indique maintenant qu'elle a lu le fichier enregistré, et le taux d'intérêt est de 7,1 même avant le code qui le change.

Voir aussi

- Sérialisation (C#)
- Guide de programmation C#

Instructions, expressions et opérateurs (guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Le code C# qui comprend une application se compose d'instructions à base de mots clés, d'expressions et d'opérateurs. Cette section contient des informations concernant ces éléments fondamentaux d'un programme C#.

Pour plus d'informations, consultez les pages suivantes :

- Instructions
- Opérateurs et expressions
- Membres expression-bodied
- Fonctions anonymes
- Comparaisons d'égalité

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Cast et conversions de types

Instructions (Guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Les actions qu'un programme effectue sont exprimées dans les instructions. Les actions courantes incluent la déclaration de variables, l'assignation de valeurs, l'appel de méthodes, l'exécution de boucles dans les collections et la création de branches sur des blocs de code, selon une condition donnée. L'ordre dans lequel les instructions sont exécutées dans un programme est appelé le flux de contrôle ou le flux d'exécution. Le flux de contrôle peut varier chaque fois qu'un programme est exécuté, selon la manière dont le programme réagit à l'entrée qu'il reçoit au moment de l'exécution.

Une instruction peut comporter une seule ligne de code terminée par un point-virgule ou une série d'instructions d'une ligne dans un bloc. Un bloc d'instructions est placé entre des accolades ({}) et peut contenir des blocs imbriqués. Le code suivant illustre deux exemples d'instructions d'une ligne, et un bloc d'instructions multiligne :

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types d'instructions

Le tableau suivant répertorie les différents types d'instructions en C# et leurs mots clés associés, avec des liens vers des rubriques contenant plus d'informations :

CATEGORY	MOTS CLÉS C# / REMARQUES
Instructions de déclaration	Une instruction de déclaration introduit une nouvelle variable ou constante. Une déclaration de variable peut éventuellement assigner une valeur à la variable. Dans une déclaration de constante, l'assignation est obligatoire.
Instructions d'expression	Les instructions d'expression qui calculent une valeur doivent stocker la valeur dans une variable.
Instructions de sélection	Les instructions de sélection permettent de créer des branches vers différentes sections de code, selon une ou plusieurs conditions spécifiées. Pour plus d'informations, voir les rubriques suivantes : • if • else • switch • études
Instructions d'itération	Les instructions d'itération permettent d'exécuter une boucle dans des collections telles que des tableaux, ou d'effectuer à plusieurs reprises le même jeu d'instructions jusqu'à ce qu'une condition spécifiée soit remplie. Pour plus d'informations, voir les rubriques suivantes : • do • for • foreach • dans • durant
Instructions de saut	Les instructions de saut transfèrent le contrôle vers une autre section de code. Pour plus d'informations, voir les rubriques suivantes : • break • pouvoir • valeurs • goto • renvoi • yield
Instructions de gestion des exceptions	Les instructions de gestion des exceptions permettent de récupérer normalement en cas de conditions exceptionnelles au moment de l'exécution. Pour plus d'informations, voir les rubriques suivantes : • lever • try-catch • try-finally • try-catch-finally

CATEGORY	MOTS CLÉS C# / REMARQUES
Activé et désactivé	Les instructions checked et unchecked vous permettent de spécifier si les opérations numériques sont autorisées à provoquer un dépassement de capacité lorsque le résultat est stocké dans une variable trop petite pour contenir la valeur résultante. Pour plus d'informations, consultez checked et unchecked.
Instruction `await`	Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression `await` dans la méthode async, il retourne à l'appelant, et la progression dans la méthode est interrompue jusqu'à ce que la tâche attendue soit terminée. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode. Pour obtenir un exemple simple, consultez la section « Méthodes async » de Méthodes. Pour plus d'informations, consultez programmation asynchrone avec Async et await.
Instruction `yield return`	Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution redémarre à partir de cet emplacement au prochain appel de l'itérateur. Pour plus d'informations, consultez itérateurs.
Instruction `fixed`	L'instruction fixed empêche le récupérateur de mémoire de déplacer une variable mobile. Pour plus d'informations, consultez fixed.
Instruction `lock`	L'instruction lock permet de limiter l'accès aux blocs de code à un seul thread à la fois. Pour plus d'informations, consultez lock.
Instructions étiquetées	Vous pouvez donner une étiquette à une instruction, puis utiliser le mot clé goto pour sauter jusqu'à l'instruction étiquetée. (Examinez l'exemple de la ligne suivante.)
Instruction vide	L'instruction vide se compose seulement d'un point-virgule. Elle ne fait rien et peut être utilisée à un emplacement où une instruction est requise alors qu'aucune action ne doit être effectuée.

Instructions de déclaration

Le code suivant montre des exemples de déclarations de variables avec et sans assignation initiale, ainsi qu'une déclaration de constante avec l'initialisation nécessaire.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Instructions d'expression

Le code suivant montre des exemples d'instructions d'expression, y compris l'assignation, la création d'objets avec assignation et l'appel de méthode.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

Instruction vide

Les exemples suivants illustrent deux utilisations d'une instruction vide :

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Instructions incorporées

Certaines instructions, y compris do, while, for et foreach, sont toujours suivies d'une instruction incorporée. Cette instruction incorporée peut être une seule instruction ou plusieurs instructions placées entre des accolades ({}) dans un bloc d'instructions. Même les instructions incorporées d'une seule ligne peuvent être placées entre des accolades ({}), comme illustré dans l'exemple suivant :

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Une instruction incorporée qui n'est pas placée entre des accolades ({}) ne peut pas être une instruction de déclaration ni une instruction étiquetée. Ceci est illustré dans l'exemple suivant :

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Placez l'instruction incorporée dans un bloc pour corriger l'erreur :

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Blocs d'instructions imbriqués

Les blocs d'instructions peuvent être imbriqués, comme illustré dans le code suivant :

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Instructions inaccessibles

Si le compilateur détermine que le flux de contrôle ne peut jamais atteindre une instruction particulière, il génère l'avertissement CS0162, comme illustré dans l'exemple suivant :

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

spécification du langage C#

Pour plus d'informations, consultez la section Instructions de la Spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Mots clés d'instructions
- Opérateurs et expressions C#

Membres expression-bodied (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Les définitions de corps d'expression vous permettent de fournir l'implémentation d'un membre sous une forme lisible et très concise. Vous pouvez utiliser une définition de corps d'expression chaque fois que la logique d'un membre pris en charge, comme une méthode ou une propriété, se compose d'une seule expression. La syntaxe générale d'une définition de corps d'expression est la suivante :

```
member => expression;
```

où *expression* est une expression valide.

La prise en charge des définitions de corps d'expression a été introduite pour les méthodes et les propriétés en lecture seule dans C# 6 et a été étendue dans C# 7.0. Vous pouvez utiliser des définitions de corps d'expression avec les membres de type listés dans le tableau suivant :

MEMBRE	PRISE EN CHARGE À COMPTER DE
Méthode	C# 6
Propriété en lecture seule	C# 6
Propriété	C# 7.0
Constructeur	C# 7.0
Finaliseur	C# 7.0
Indexeur	C# 7.0

Méthodes

Une méthode expression-bodied se compose d'une seule expression qui retourne une valeur dont le type correspond au type de retour de la méthode ou, pour les méthodes qui retournent `void`, qui effectue une opération. Par exemple, les types qui substituent la méthode ToString incluent généralement une expression unique qui retourne la représentation sous forme de chaîne de l'objet actuel.

L'exemple suivant définit une classe `Person` qui substitue la méthode ToString avec une définition de corps d'expression. Il définit également une méthode `DisplayName` qui affiche un nom sur la console. Notez que le mot clé `return` n'est pas utilisé dans la définition de corps d'expression `ToString`.

```
using System;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

Pour plus d'informations, consultez Méthodes (Guide de programmation C#).

Propriétés en lecture seule

Depuis C# 6, vous pouvez utiliser une définition de corps d'expression pour implémenter une propriété en lecture seule. Pour ce faire, utilisez la syntaxe suivante :

```
PropertyType PropertyName => expression;
```

L'exemple suivant définit une classe `Location` dont la propriété en lecture seule `Name` est implémentée comme une définition de corps d'expression qui retourne la valeur du champ privé `locationName` :

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

Pour plus d'informations sur les propriétés, consultez Propriétés (Guide de programmation C#).

Propriétés

Depuis C# 7.0, vous pouvez utiliser des définitions de corps d'expression pour implémenter la propriété `get` et les accesseurs `set`. L'exemple suivant montre comment faire :

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Pour plus d'informations sur les propriétés, consultez Propriétés (Guide de programmation C#).

Constructeurs

Une définition de corps d'expression pour un constructeur se compose généralement d'une seule expression d'assignation ou d'un appel de méthode qui gère les arguments du constructeur ou qui initialise l'état de l'instance.

L'exemple suivant définit une classe `Location` dont le constructeur a un seul paramètre de chaîne nommé *name*. La définition de corps d'expression assigne l'argument à la propriété `Name`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Pour plus d'informations, consultez Constructeurs (Guide de programmation C#).

Finaliseurs

Une définition de corps d'expression pour un finaliseur contient généralement des instructions de nettoyage, telles que des instructions qui libèrent les ressources non managées.

L'exemple suivant définit un finaliseur qui utilise une définition de corps d'expression pour indiquer que le finaliseur a été appelé.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

Pour plus d'informations, consultez Finaliseurs (Guide de programmation C#).

Indexeurs

Comme avec les propriétés, l'indexeur `get` et les `set` accesseurs se composent de définitions de corps d'expression si l' `get` accesseur se compose d'une expression unique qui retourne une valeur ou l' `set` accesseur effectue une assignation simple.

L'exemple suivant définit une classe nommée `Sports` qui inclut un tableau String interne contenant les noms de plusieurs sports. L'indexeur `get` et les `set` accesseurs sont tous deux implémentés en tant que définitions de corps d'expression.

```
using System;
using System.Collections.Generic;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

Pour plus d'informations, consultez Indexeurs (Guide de programmation C#).

Fonctions anonymes (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Une fonction anonyme est une instruction ou expression « inline » qui peut être utilisée partout où un type délégué est attendu. Vous pouvez l'utiliser pour initialiser un délégué nommé ou la passer à la place d'un type délégué nommé comme paramètre de méthode.

Vous pouvez utiliser une expression lambda ou une méthode anonyme pour créer une fonction anonyme. Nous vous recommandons d'utiliser des expressions lambda, car elles fournissent un moyen plus concis et plus expressif d'écrire du code en ligne. Contrairement aux méthodes anonymes, certains types d'expressions lambda peuvent être convertis en types d'arborescence d'expression.

Évolution des délégués en C#

Dans C# 1.0, vous pouviez créer une instance d'un délégué en l'initialisant explicitement avec une méthode déjà définie à un autre endroit dans le code. C# 2.0 a introduit le concept des méthodes anonymes, qui vous permettent d'écrire des blocs d'instructions inline sans nom pouvant être exécutés dans un appel de délégué. C# 3.0 a introduit les expressions lambda, qui sont semblables aux méthodes anonymes d'un point de vue conceptuel, mais qui sont plus expressives et concises. Ces deux fonctionnalités sont désignées collectivement comme des *fonctions anonymes*. En général, les applications qui ciblent .NET Framework 3,5 ou une version ultérieure doivent utiliser des expressions lambda.

L'exemple suivant montre l'évolution de la création de délégués entre C# 1.0 et C# 3.0 :

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Voir aussi

- Instructions, expressions et opérateurs
- Expressions lambda
- Délégués
- Arborescences d'expressions (C#)

Guide pratique pour utiliser des expressions lambda dans une requête (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Vous n'utilisez pas d'expressions lambda directement dans la syntaxe de requête, mais vous les utilisez dans les appels de méthode qui peuvent être contenus dans des expressions de requête. En effet, certaines opérations de requête ne peuvent être exprimées que dans une syntaxe de méthode. Pour plus d'informations sur la différence entre la syntaxe de requête et la syntaxe de méthode, consultez Syntaxe de requête et syntaxe de méthode dans LINQ.

Exemple

L'exemple suivant montre comment utiliser une expression lambda dans une requête fondée sur une méthode à l'aide de l'opérateur de requête standard Enumerable.Where. Notez que la méthode Where de cet exemple a un paramètre d'entrée du type délégué Func<T,TResult>, et que ce délégué prend un entier comme entrée et retourne une valeur booléenne. L'expression lambda peut être convertie en ce délégué. S'il s'agissait d'une requête LINQ to SQL utilisant la méthode Queryable.Where, le type de paramètre serait un `Expression<Func<int,bool>>`, mais l'expression lambda serait exactement la même. Pour plus d'informations sur le type Expression, consultez System.Linq.Expressions.Expression.

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

Exemple

L'exemple suivant montre comment utiliser une expression lambda dans un appel de méthode d'une expression de requête. L'expression lambda est nécessaire, car l'opérateur de requête standard Sum ne peut pas être appelé à l'aide de la syntaxe de requête.

La requête regroupe d'abord les étudiants selon leurs notes, comme défini dans l'enum `GradeLevel`. Ensuite, pour chaque groupe, elle ajoute la note totale obtenue par chaque étudiant. Cela nécessite deux opérations `Sum`. Le `Sum` interne calcule la note totale obtenue par chaque étudiant, et le `Sum` externe obtient un total combiné de tous les étudiants du groupe.

```
private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
    from student in students
    group student by student.Year into studentGroup
    select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
     Outputs:
     Key = SecondYear Sum = 1014
     Key = ThirdYear Sum = 964
     Key = FirstYear Sum = 1058
     Key = FourthYear Sum = 974
*/
```

Compilation du code

Pour exécuter ce code, copiez et collez la méthode dans le `StudentClass` fourni dans interroger une collection d'objets et appelez-la à partir de la `Main` méthode.

Voir aussi

- Expressions lambda
- Arborescences d'expressions (C#)

Comparaisons d'égalité (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Il est parfois nécessaire de comparer l'égalité de deux valeurs. Dans certains cas, vous testez *l'égalité de valeur*, également appelée *équivalence*, qui signifie que les valeurs contenues dans les deux variables sont égales. Dans d'autres cas, vous devez déterminer si deux variables référencent le même objet sous-jacent en mémoire. Ce type d'égalité est appelée *l'égalité de référence* ou *identité*. Cette rubrique décrit ces deux genres d'égalité et fournit des liens vers d'autres rubriques pour obtenir plus d'informations.

Égalité des références

L'égalité de références signifie que deux références d'objets référencent le même objet sous-jacent. Cela peut se produire par simple assignation, comme illustré dans l'exemple suivant.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Dans ce code, deux objets sont créés, mais après l'instruction d'assignation, les deux références référencent le même objet. Elles présentent donc une égalité de référence. Utilisez la méthode ReferenceEquals pour déterminer si deux références référencent le même objet.

Le concept d'égalité de référence s'applique uniquement aux types référence. Les objets de type valeur ne peuvent pas avoir une égalité de référence, car quand une instance d'un type valeur est assignée à une variable, une copie de la valeur est effectuée. Ainsi, vous ne pouvez jamais avoir deux structs unboxed qui référencent le même emplacement en mémoire. De plus, si vous utilisez ReferenceEquals pour comparer deux types valeur, le résultat est toujours `false`, même si les valeurs contenues dans les objets sont toutes identiques. Cela est dû au fait que chaque variable est convertie en une instance d'objet distincte. Pour plus d'informations, consultez Comment tester

l'égalité des références (identité).

Égalité des valeurs

L'égalité de valeur signifie que deux objets contiennent les mêmes valeurs. Pour les types valeur primitifs tels que int ou bool, les tests d'égalité de valeur sont simples. Vous pouvez utiliser l' `==` opérateur, comme indiqué dans l'exemple suivant.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

Pour la plupart des autres types, les tests d'égalité de valeur sont plus complexes parce qu'ils vous demandent de comprendre comment le type la définit. Pour les classes et structs qui ont plusieurs champs ou propriétés, l'égalité de valeur signifie souvent que tous les champs ou propriétés ont la même valeur. Par exemple, deux objets `Point` peuvent être définis comme équivalents si pointA.X est égal à pointB.X et que pointA.Y est égal à pointB.Y.

Toutefois, il n'est pas obligatoire que l'équivalence soit basée sur tous les champs dans un type. Elle peut être basée sur une partie. Quand vous comparez des types dont vous n'êtes pas propriétaire, vous devez bien comprendre comment l'équivalence est définie pour ce type. Pour plus d'informations sur la définition de l'égalité des valeurs dans vos propres classes et structs, consultez comment définir l'égalité des valeurs pour un type.

Égalité des valeurs à virgule flottante

Les comparaisons d'égalité des valeurs à virgule flottante (double et float) sont problématiques en raison de l'imprécision de l'arithmétique à virgule flottante sur les ordinateurs binaires. Pour plus d'informations, consultez les notes dans la rubrique System.Double.

Rubriques connexes

TITRE	DESCRIPTION
Comment tester l'égalité des références (identité)	Décrit comment déterminer si deux variables présentent une égalité de référence.
Comment définir une égalité de valeurs pour un type	Décrit comment fournir une définition personnalisée de l'égalité de valeur pour un type.
Guide de programmation C#	Fournit des liens vers des informations détaillées sur les fonctionnalités et fonctionnalités importantes du langage C# qui sont disponibles pour C# via .NET.
Types	Fournit des informations sur le système de typeC# et des liens vers des informations supplémentaires.

Voir aussi

- Guide de programmation C#

Comment définir l'égalité des valeurs pour un type (Guide de programmation C#)

02/11/2020 • 12 minutes to read • Edit Online

Quand vous définissez une classe ou un struct, vous décidez s'il est judicieux de créer une définition personnalisée de l'égalité des valeurs (ou équivalence) pour le type. En général, vous implémentez l'égalité des valeurs quand des objets du type sont supposés être ajoutés à une collection quelconque, ou quand leur objectif principal est de stocker un ensemble de champs ou propriétés. Vous pouvez baser votre définition de l'égalité des valeurs sur une comparaison de tous les champs et propriétés du type, ou vous pouvez la baser sur un sous-ensemble.

Dans les deux cas, et dans les classes et les structs, votre implémentation doit suivre les cinq garanties d'équivalence (pour les règles suivantes, supposez que `x` , `y` et ne `z` sont pas null) :

1. `x.Equals(x)` retourne `true`. Il s'agit de la propriété réflexive.
2. `x.Equals(y)` retourne la même valeur que `y.Equals(x)`. Il s'agit de la propriété symétrique.
3. si `(x.Equals(y) && y.Equals(z))` retourne `true`, `x.Equals(z)` retourne `true`. Il s'agit de la propriété transitive.
4. Les appels successifs de `x.Equals(y)` retournent la même valeur tant que les objets référencés par x et y ne sont pas modifiés.
5. Toute valeur non NULL n'est pas égale à null. Toutefois, le CLR vérifie la valeur NULL sur tous les appels de méthode et lève une `NullReferenceException` si la `this` référence aurait la valeur null. Par conséquent, `x.Equals(y)` lève une exception lorsque `x` a la valeur null. Cela arrête les règles 1 ou 2, en fonction de l'argument de `Equals` .

Tout struct que vous définissez a déjà une implémentation par défaut de l'égalité des valeurs dont il hérite de la substitution System.ValueType de la méthode Object.Equals(Object). Cette implémentation utilise la réflexion pour examiner tous les champs et propriétés du type. Bien que cette implémentation produise des résultats corrects, elle est relativement lente par rapport à une implémentation personnalisée que vous écrivez spécifiquement pour le type.

Les détails d'implémentation pour l'égalité des valeurs sont différents pour les classes et les structs. Toutefois, les classes et les structs nécessitent tous deux les mêmes étapes de base pour l'implémentation de l'égalité :

1. Substituez la méthode virtual Object.Equals(Object). Dans la plupart des cas, votre implémentation de `bool Equals( object obj )` doit simplement appeler la méthode `Equals` propre au type qui est l'implémentation de l'interface System.IEquatable<T>. (Voir l'étape 2.)
2. Implémentez l'interface System.IEquatable<T> en fournissant une méthode `Equals` propre au type. C'est ici qu'est effectuée la comparaison d'équivalence. Par exemple, vous pouvez décider de définir l'égalité en comparant seulement un ou deux champs de votre type. Ne levez pas d'exceptions à partir de la méthode `Equals`. Pour les classes uniquement : cette méthode doit examiner uniquement les champs qui sont déclarés dans la classe. Elle doit appeler `base.Equals` pour examiner les champs qui sont dans la classe de base. (Ne faites pas cela si le type hérite directement d'Object, car l'implémentation Object de Object.Equals(Object) effectue une vérification de l'égalité de référence.)
3. Facultatif mais recommandé : Surchargez les `==` opérateurs et `!=` .
4. Substituez Object.GetHashCode pour que deux objets ayant une égalité des valeurs produisent le même

code de hachage.

5. Facultatif : pour prendre en charge les définitions pour « supérieur à » ou « inférieur à », implémentez l' IComparable<T> interface pour votre type, et surchargez également les <= >= opérateurs et.

> **NOTE**
> À compter de C# 9,0, vous pouvez utiliser des enregistrements pour récupérer la sémantique d'égalité des valeurs sans code réutilisable inutile.

Exemple de classe

L'exemple suivant montre comment implémenter l'égalité des valeurs dans une classe (type référence).

```
namespace ValueEquality
{
    using System;
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        // Readonly auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        // Set the properties in the constructor.
        public TwoDPoint(int x, int y)
        {
            if ((x < 1) || (x > 2000) || (y < 1) || (y > 2000))
            {
                throw new System.ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj)
        {
            return this.Equals(obj as TwoDPoint);
        }

        public bool Equals(TwoDPoint p)
        {
            // If parameter is null, return false.
            if (Object.ReferenceEquals(p, null))
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }
```

```
    public override int GetHashCode()
    {
        return X * 0x00010000 + Y;
    }

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        // Check for null on left side.
        if (Object.ReferenceEquals(lhs, null))
        {
            if (Object.ReferenceEquals(rhs, null))
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs == rhs);
    }
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new System.ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj)
    {
        return this.Equals(obj as ThreeDPoint);
    }

    public bool Equals(ThreeDPoint p)
    {
        // If parameter is null, return false.
        if (Object.ReferenceEquals(p, null))
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
```

```
                return base.Equals((TwoDPoint)p);
            }
            else
            {
                return false;
            }
        }

        public override int GetHashCode()
        {
            return (X * 0x100000) + (Y * 0x1000) + Z;
        }

        public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            // Check for null.
            if (Object.ReferenceEquals(lhs, null))
            {
                if (Object.ReferenceEquals(rhs, null))
                {
                    // null == null = true.
                    return true;
                }

                // Only the left side is null.
                return false;
            }
            // Equals handles the case of null on right side.
            return lhs.Equals(rhs);
        }

        public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            return !(lhs == rhs);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointC = null;
            int i = 5;

            Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
            Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

            TwoDPoint pointD = null;
            TwoDPoint pointE = null;

            Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

            pointE = new TwoDPoint(3, 4);
            Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
            Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
            Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

            System.Collections.ArrayList list = new System.Collections.ArrayList();
            list.Add(new ThreeDPoint(3, 4, 5));
            Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
```

```
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        null comparison = False
        Compare to some other type = False
        Two null TwoDPoints are equal: True
        (pointE == pointA) = False
        (pointA == pointE) = False
        (pointA != pointE) = True
        pointE.Equals(list[0]): False
    */
}
```

Sur les classes (types référence), l'implémentation par défaut des deux méthodes Object.Equals(Object) effectue une comparaison d'égalité de référence, et non une vérification de l'égalité des valeurs. Quand un implémenteur substitue la méthode virtuelle, l'objectif est de lui donner une sémantique d'égalité des valeurs.

Les opérateurs `==` et `!=` peuvent être utilisés avec des classes, même si la classe ne les surcharge pas. Toutefois, le comportement par défaut consiste à effectuer une vérification de l'égalité de référence. Dans une classe, si vous surchargez la méthode `Equals`, vous devez surcharger les opérateurs `==` et `!=`, mais cela n'est pas obligatoire.

Exemple de struct

L'exemple suivant montre comment implémenter l'égalité des valeurs dans un struct (type valeur) :

```
using System;
struct TwoDPoint : IEquatable<TwoDPoint>
{
    // Read/write auto-implemented properties.
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
        : this()
    {
        X = x;
        Y = x;
    }

    public override bool Equals(object obj)
    {
        if (obj is TwoDPoint)
        {
            return this.Equals((TwoDPoint)obj);
        }
        return false;
    }

    public bool Equals(TwoDPoint p)
    {
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode()
    {
        return X ^ Y;
    }

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        return lhs.Equals(rhs);
    }
```

```
    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs.Equals(rhs));
    }
}

class Program
{
    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // Compare using virtual Equals, static Equals, and == and != operators.
        // True:
        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        // True:
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        // True:
        Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));
        // False:
        Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
        // False:
        Console.WriteLine("(pointA == null) = {0}", pointA == null);
        // True:
        Console.WriteLine("(pointA != null) = {0}", pointA != null);
        // False:
        Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
        // CS0019:
        // Console.WriteLine("pointA == i = {0}", pointA == i);

        // Compare unboxed to boxed.
        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new TwoDPoint(3, 4));
        // True:
        Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

        // Compare nullable to nullable and to non-nullable.
        TwoDPoint? pointC = null;
        TwoDPoint? pointD = null;
        // False:
        Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
        // True:
        Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

        TwoDPoint temp = new TwoDPoint(3, 4);
        pointC = temp;
        // True:
        Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

        pointD = temp;
        // True:
        Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    Object.Equals(pointA, pointB) = True
    pointA.Equals(null) = False
    (pointA == null) = False
    (pointA != null) = True
    pointA.Equals(i) = False
```

```
        pointA.Equals(1) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

Pour les structs, l'implémentation par défaut de Object.Equals(Object) (qui est la version substituée dans System.ValueType) effectue une vérification de l'égalité des valeurs à l'aide de la réflexion pour comparer les valeurs de chaque champ dans le type. Quand un implémenteur substitue la méthode `Equals` virtuelle dans un struct, l'objectif est de fournir un moyen plus efficace d'effectuer la vérification de l'égalité des valeurs, et éventuellement de baser la comparaison sur un sous-ensemble des propriétés ou du champ du struct.

Les `==` opérateurs et `!=` ne peuvent pas fonctionner sur un struct, sauf si le struct les surcharge explicitement.

Voir aussi

- Comparaisons d'égalité
- Guide de programmation C#

Comment tester l'égalité des références (identité) (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Il n'est pas utile d'implémenter une logique personnalisée pour prendre en charge les comparaisons d'égalité de références dans vos types. Cette fonctionnalité est fournie pour tous les types par la méthode Object.ReferenceEquals statique.

L'exemple suivant montre comment déterminer si deux variables affichent une *égalité de références*, à savoir qu'elles font référence à un même objet en mémoire.

L'exemple montre aussi pourquoi Object.ReferenceEquals retourne toujours `false` pour les types valeur et pourquoi vous ne devez pas utiliser ReferenceEquals pour déterminer l'égalité de chaînes.

Exemple

```
namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
```

```
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) = {0}",
                              Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned by the runtime.
            // This means they are stored in the same location in memory. Therefore,
            // the two strings have reference equality although no assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                              Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                              Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                              Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
*/
```

L'implémentation de `Equals` dans la classe de base universelle System.Object assure aussi une vérification de l'égalité de références, mais il est préférable de ne pas l'utiliser, car si une classe se substitue à la méthode, vous risquez d'obtenir des résultats inattendus. Il en va de même pour les opérateurs `==` et `!=`. Quand ils s'appliquent à des types référence, le comportement par défaut de `==` et de `!=` est d'effectuer une vérification de l'égalité des

références. Cependant, les classes dérivées peuvent surcharger l'opérateur pour effectuer une vérification d'égalité de valeurs. Pour réduire le risque d'erreurs, il est recommandé d'utiliser systématiquement ReferenceEquals quand il s'agit de déterminer si deux objets présentent une égalité de références.

Les chaînes constantes au sein d'un même assembly sont toujours intégrées par le runtime. Autrement dit, une seule instance de chaque chaîne littérale unique est conservée. En revanche, le runtime ne garantit pas que les chaînes créées au moment de l'exécution sont intégrées, ni que deux chaînes constantes égales d'assemblys différents sont intégrées.

Voir aussi

- Comparaisons d'égalité

Types (Guide de programmation C#)

16/12/2020 • 25 minutes to read • Edit Online

Types, variables et valeurs

C# est un langage fortement typé. Chaque variable et chaque constante ont un type, tout comme chaque expression qui fournit une valeur. Chaque déclaration de méthode spécifie un nom, un nombre de paramètres, ainsi que le type et le type (valeur, référence ou sortie) pour chaque paramètre d'entrée et pour la valeur de retour. La bibliothèque de classes .NET définit un ensemble de types numériques intégrés et des types plus complexes qui représentent un large éventail de constructions logiques, telles que le système de fichiers, les connexions réseau, les collections et les tableaux d'objets, ainsi que les dates. Un programme C# classique utilise des types de la bibliothèque de classes et des types définis par l'utilisateur qui modélisent les concepts propres au domaine du problème du programme.

Les informations stockées dans un type peuvent inclure les éléments suivants :

- Espace de stockage nécessaire pour une variable du type.
- Valeurs minimale et maximale que le type peut représenter.
- Membres (méthodes, champs, événements, etc.) que le type contient.
- Type de base dont le type est hérité.
- Interface (s) qu'il implémente.
- Emplacement où sera allouée la mémoire pour les variables au moment de l'exécution.
- Sortes d'opérations autorisées.

Le compilateur utilise les informations de type pour s'assurer que toutes les opérations effectuées dans votre code sont de *type sécurisé*. Par exemple, si vous déclarez une variable de type int, le compilateur vous permet d'utiliser la variable dans des opérations d'addition et de soustraction. Si vous essayez d'effectuer ces mêmes opérations avec une variable de type bool, le compilateur génère une erreur, comme illustré dans l'exemple suivant :

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> Développeurs C et C++ : notez que dans C#, bool n'est pas convertible en int.

Le compilateur incorpore les informations de type dans le fichier exécutable sous forme de métadonnées. Le common language runtime (CLR) utilise ces métadonnées au moment de l'exécution pour garantir que le type est sécurisé lorsqu'il alloue et libère de la mémoire.

Spécification de types dans les déclarations de variable

Lorsque vous déclarez une variable ou une constante dans un programme, vous devez spécifier son type ou utiliser le mot clé var pour permettre au compilateur de déduire le type. L'exemple suivant montre des déclarations de variable qui utilisent des types numériques intégrés et des types complexes définis par

l'utilisateur :

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Les types de paramètres de méthode et de valeurs de retour sont spécifiés dans la déclaration de méthode. La signature suivante présente une méthode qui requiert int comme argument d'entrée et retourne une chaîne :

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Une fois que vous avez déclaré une variable, vous ne pouvez pas la redéclarer avec un nouveau type et vous ne pouvez pas assigner une valeur qui n'est pas compatible avec son type déclaré. Par exemple, vous ne pouvez pas déclarer un int et lui assigner une valeur booléenne `true` . Toutefois, les valeurs peuvent être converties en d'autres types, par exemple quand elles sont assignées à de nouvelles variables ou passées en tant qu'arguments de méthode. Une *conversion de type* qui n'entraîne pas de perte de données est effectuée automatiquement par le compilateur. Une conversion susceptible de causer la perte de données exige une variable *cast* dans le code source.

Pour plus d'informations, consultez Cast et conversions de types.

Types intégrés

C# fournit un jeu standard de types intégrés pour représenter les entiers, les valeurs à virgule flottante, les expressions booléennes, les caractères de texte, les valeurs décimales et d'autres types de données. Il existe également des types `string` et `object` intégrés. Ces types peuvent être utilisés dans n'importe quel programme C#. Pour obtenir la liste complète des types intégrés, consultez types intégrés.

Types personnalisés

Vous utilisez les constructions struct, classe, interface et enum pour créer vos propres types personnalisés. La bibliothèque de classes .NET est une collection de types personnalisés fournie par Microsoft que vous pouvez utiliser dans vos propres applications. Par défaut, les types les plus fréquemment utilisés dans la bibliothèque de classes sont disponibles dans tous les programmes C#. D'autres deviennent disponibles uniquement lorsque vous ajoutez explicitement une référence de projet à l'assembly dans lequel elles sont définies. Une fois que le compilateur a une référence à l'assembly, vous pouvez déclarer des variables (et des constantes) des types déclarés dans cet assembly dans le code source. Pour plus d'informations, consultez Bibliothèque de classes .NET.

Système de type commun

Il est important de comprendre deux points fondamentaux concernant le système de type dans .NET :

- Il prend en charge le principe d'héritage. Les types peuvent dériver d'autres types, appelés *types de base*. Le type dérivé hérite (avec certaines restrictions) des méthodes, des propriétés et des autres membres du type de base. Le type de base peut à son tour dériver d'un autre type, auquel cas le type dérivé hérite des membres des deux types de base dans sa hiérarchie d'héritage. Tous les types, notamment les types numériques intégrés comme System.Int32 (mot clé C# : int), dérivent au final d'un seul type de base, qui est System.Object (mot clé C# : object). Cette hiérarchie de types unifiée est appelée Système de type commun (CTS). Pour plus d'informations sur l'héritage dans C#, consultez Héritage.
- Chaque type du CTS est défini comme *type valeur* ou *type référence*. Ces types incluent tous les types personnalisés dans la bibliothèque de classes .NET et également vos propres types définis par l'utilisateur. Les types que vous définissez à l'aide du mot clé struct sont des types valeur ; tous les types numériques intégrés sont `structs`. Les types que vous définissez à l'aide du mot clé class sont des types référence. Les types référence et les types valeur ne suivent pas les mêmes règles de compilation et ont un comportement différent au moment de l'exécution.

L'illustration suivante montre la relation entre les types valeur et les types référence dans le CTS.

L'image suivante montre les types valeur et les types référence dans CTS :

> **NOTE**
>
> Vous pouvez voir que les types couramment utilisés sont tous organisés dans l'espace de noms System. Toutefois, l'espace de noms qui contient un type n'a aucune incidence sur le fait qu'il s'agisse d'un type valeur ou d'un type référence.

Types de valeur

Les types valeur dérivent de System.ValueType, qui dérive de System.Object. Les types qui dérivent de System.ValueType ont un comportement spécial dans le CLR. Les variables de type valeur contiennent directement leurs valeurs, ce qui signifie que la mémoire est allouée inline dans le contexte où la variable est déclarée. Il n'existe pas d'allocation de tas ou de charge de garbage collection distincte pour les variables de type valeur.

Il existe deux catégories de types valeur : struct et enum.

Les types numériques intégrés sont des structs et ont des champs et des méthodes auxquels vous pouvez accéder :

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Mais vous les déclarez et leur assignez des valeurs comme s'il s'agissait de types non agrégés simples :

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Les types valeur sont *sealed*, ce qui signifie que vous ne pouvez pas dériver un type de n'importe quel type valeur, par exemple System.Int32 . Vous ne pouvez pas définir un struct pour hériter d'une classe ou d'un struct défini par l'utilisateur, car un struct peut uniquement hériter de System.ValueType . Toutefois, un struct peut implémenter une ou plusieurs interfaces. Vous pouvez effectuer un cast d'un type struct en n'importe quel type d'interface qu'il implémente ; ce Cast entraîne une opération de *boxing* pour encapsuler le struct à l'intérieur d'un objet de type référence sur le tas managé. Les opérations de boxing se produisent quand vous passez un type valeur à une méthode qui prend un System.Object ou tout type d'interface comme paramètre d'entrée. Pour plus d'informations, consultez Conversion boxing et unboxing.

Vous utilisez le mot clé struct pour créer vos propres types valeur personnalisés. En règle générale, un struct est utilisé comme conteneur pour un petit jeu de variables connexes, comme le montre l'exemple suivant :

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Pour plus d'informations sur les structs, consultez types structure. Pour plus d'informations sur les types valeur, consultez types valeur.

L'autre catégorie de types valeur est enum. Un enum définit un jeu de constantes intégrales nommées. Par exemple l'énumération System.IO.FileMode dans la bibliothèque de classes du .NET contient un ensemble d'entiers constants nommés qui spécifient comment un fichier doit être ouvert. Il est défini comme indiqué dans l'exemple suivant :

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

La constante `System.IO.FileMode.Create` a la valeur 2. Toutefois, le nom est bien plus explicite pour les êtres humains qui lisent le code source. pour cette raison, il est préférable d'utiliser des énumérations au lieu de nombres littéraux constants. Pour plus d'informations, consultez System.IO.FileMode.

Toutes les énumérations héritent de System.Enum, qui hérite de System.ValueType. Toutes les règles qui s'appliquent aux structs s'appliquent également aux enums. Pour plus d'informations sur les enums, consultez

types énumération.

Types référence

Un type qui est défini comme une classe, un délégué, un tableau ou une interface est un *type référence*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur null tant que vous n'avez pas explicitement créé un objet à l'aide de l'opérateur new ou que vous ne lui avez pas assigné un objet créé ailleurs à l'aide de `new`, comme indiqué dans l'exemple suivant :

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

Une interface doit être initialisée avec un objet de classe qui l'implémente. Si `MyClass` implémente `IMyInterface`, vous créez une instance de `IMyInterface` comme indiqué dans l'exemple suivant :

```
IMyInterface iface = new MyClass();
```

Quand l'objet est créé, la mémoire est allouée sur le tas managé et la variable contient uniquement une référence à l'emplacement de l'objet. Les types sur le tas managé nécessitent une surcharge lorsqu'ils sont alloués et lorsqu'ils sont récupérés par la fonctionnalité de gestion automatique de la mémoire du CLR, appelée *garbage collection*. Toutefois, garbage collection est également hautement optimisé et, dans la plupart des scénarios, il ne crée pas de problème de performances. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire.

Tous les tableaux sont des types référence, même si leurs éléments sont des types valeur. Les tableaux dérivent implicitement de la classe System.Array, mais vous les déclarez et vous les utilisez avec la syntaxe simplifiée fournie par C#, comme illustré dans l'exemple suivant :

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Les types référence prennent en charge l'héritage. Lorsque vous créez une classe, vous pouvez hériter de toute autre interface ou classe qui n'est pas définie comme sealed, et d'autres classes peuvent hériter de votre classe et substituer vos méthodes virtuelles. Pour plus d'informations sur la création de vos propres classes, consultez Classes et structures. Pour plus d'informations sur l'héritage et les méthodes virtuelles, consultez Héritage.

Types de valeurs littérales

Dans C#, les valeurs littérales reçoivent un type du compilateur. Vous pouvez spécifier la façon dont un littéral numérique doit être typé en ajoutant une lettre à la fin du nombre. Par exemple, pour spécifier que la valeur 4,56 doit être traitée comme une valeur float, ajoutez « f » ou « F » après le nombre : `4.56f`. Si aucune lettre n'est ajoutée, le compilateur déduit un type pour le littéral. Pour plus d'informations sur les types qui peuvent être spécifiés avec des suffixes de lettres, consultez types numériques intégraux et types numériques à virgule flottante.

Étant donné que les littéraux sont typés et que tous les types dérivent en fin de compte System.Object , vous pouvez écrire et compiler du code tel que le code suivant :

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Types génériques

Un type peut être déclaré avec un ou plusieurs *paramètres de type* qui servent d'espace réservé pour le type réel (le *type concret*) que le code client fournit lorsqu'il crée une instance du type. Ces types sont appelés *types génériques*. Par exemple, le type .NET System.Collections.Generic.List<T> a un paramètre de type qui, par Convention, porte le nom *T*. Lorsque vous créez une instance du type, vous spécifiez le type des objets que la liste doit contenir, par exemple String :

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

L'utilisation du paramètre de type rend possible la réutilisation de la même classe pour contenir tout type d'élément, sans avoir à convertir chaque élément en object. Les classes de collection génériques sont appelées *collections fortement typées* , car le compilateur connaît le type spécifique des éléments de la collection et peut déclencher une erreur au moment de la compilation si, par exemple, vous tentez d'ajouter un entier à l' `stringList` objet dans l'exemple précédent. Pour plus d'informations, consultez Génériques.

Types implicites, types anonymes et types valeur Nullable

Comme indiqué précédemment, vous pouvez attribuer implicitement un type à une variable locale (mais pas les membres de la classe) à l'aide du mot clé var. La variable reçoit toujours un type au moment de la compilation, mais le type est fourni par le compilateur. Pour plus d'informations, consultez Variables locales implicitement typées.

Il peut être difficile de créer un type nommé pour des ensembles simples de valeurs associées que vous ne souhaitez pas stocker ou passer en dehors des limites de la méthode. Vous pouvez alors créer des *types anonymes*. Pour plus d'informations, consultez types anonymes.

Les types valeur ordinaires ne peuvent pas avoir une valeur null. Toutefois, vous pouvez créer des types valeur Nullable en ajoutant un `?` après le type. Par exemple, `int?` est un type `int` qui peut également avoir la valeur null. Les types valeur Nullable sont des instances du type struct générique System.Nullable<T> . Les types valeur Nullable sont particulièrement utiles lorsque vous passez des données vers et à partir de bases de données dans lesquelles les valeurs numériques peuvent être null. Pour plus d'informations, consultez types valeur Nullable.

Type au moment de la compilation et type d'exécution

Une variable peut avoir différents types au moment de la compilation et au moment de l'exécution. Le *type au moment* de la compilation est le type déclaré ou déduit de la variable dans le code source. Le *type au moment* de l'exécution est le type de l'instance référencée par cette variable. Ces deux types sont souvent les mêmes, comme dans l'exemple suivant :

```
string message = "This is a string of characters";
```

Dans d'autres cas, le type au moment de la compilation est différent, comme indiqué dans les deux exemples suivants :

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

Dans les deux exemples précédents, le type au moment de l'exécution est un `string` . Le type au moment de la compilation est `object` sur la première ligne et `IEnumerable<char>` dans la seconde.

Si les deux types sont différents pour une variable, il est important de comprendre quand le type au moment de la compilation et le type au moment de l'exécution s'appliquent. Le type au moment de la compilation détermine toutes les actions effectuées par le compilateur. Ces actions du compilateur incluent la résolution de l'appel de méthode, la résolution de surcharge et les casts implicites et explicites disponibles. Le type au moment de l'exécution détermine toutes les actions qui sont résolues au moment de l'exécution. Ces actions au moment de l'exécution incluent la répartition des appels de méthode virtuelle, l'évaluation `is` et les `switch` expressions, ainsi que d'autres API de test de type. Pour mieux comprendre comment votre code interagit avec les types, identifiez l'action qui s'applique à ce type.

Sections connexes

Pour plus d'informations, consultez les articles suivants :

- Cast et conversions de types
- Conversion boxing et unboxing
- Utilisation du type dynamic
- Types valeur
- Types référence
- Classes et structs
- Types anonymes
- Génériques

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Conversion des types de données XML
- Types intégraux

Cast et conversions de types (guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

C# étant typé statiquement au moment de la compilation, une fois qu'une variable est déclarée elle ne peut plus être redéclarée et aucune valeur d'un autre type ne peut lui être assignée, sauf si ce type peut être converti de manière implicite au type de la variable. Par exemple, `string` ne peut pas être converti implicitement en `int`. Après avoir déclaré `i` comme `int`, vous ne pouvez donc pas lui assigner la chaîne « Hello », comme l'illustre le code suivant :

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

Cependant, vous pouvez être amené à copier une valeur dans un paramètre de variable ou de méthode d'un autre type. C'est notamment le cas si vous avez une variable de type entier que vous devez passer à une méthode dont le paramètre est de type `double`. Il peut également arriver que vous deviez assigner une variable de classe à une variable de type interface. Les opérations de ce genre sont appelées « *conversions de types* ». En C#, vous pouvez effectuer les conversions suivantes :

- **Conversions implicites**: aucune syntaxe spéciale n'est requise, car la conversion échoue toujours et aucune donnée n'est perdue. Citons par exemple les conversions de types intégraux en d'autres plus importants, et les conversions de classes dérivées en classes de base.
- **Conversions explicites (CASTS)**: les conversions explicites requièrent une expression de cast. Un cast est exigé quand les informations peuvent être perdues durant la conversion, ou quand la conversion peut échouer pour d'autres raisons. Exemples classiques : conversion numérique en type qui a moins de précision ou une plus petite plage, et conversion d'une instance de classe de base en classe dérivée.
- **Conversions définies par l'utilisateur** : les conversions définies par l'utilisateur sont effectuées par des méthodes spéciales que vous pouvez définir pour permettre des conversions explicites ou implicites entre des types personnalisés qui n'ont pas de relation classe de base/classe dérivée. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.
- **Conversions avec les classes d'assistance** : pour effectuer une conversion entre des types non compatibles, tels que des entiers et des objets System.DateTime ou des chaînes hexadécimales et des tableaux d'octets, vous pouvez utiliser la classe System.BitConverter, la classe System.Convert et les méthodes `Parse` des types numériques intégrés, comme Int32.Parse. Pour plus d'informations, consultez Comment convertir un tableau d'octets en int, Comment convertir une chaîne en nombreet Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques.

Conversions implicites

Pour les types numériques intégrés, une conversion implicite peut être effectuée quand la valeur à stocker peut tenir dans la variable sans être tronquée ni arrondie. Pour les types intégraux, cela signifie que la plage du type de source est un sous-ensemble approprié de la plage du type cible. Par exemple, une variable de type long (entier sur 64 bits) peut stocker n'importe quelle valeur pouvant être stockée par un int (entier sur 32 bits). Dans l'exemple suivant, le compilateur convertit implicitement la valeur de `num` à droite en type `long` avant de l'assigner à `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Pour obtenir la liste complète de toutes les conversions numériques implicites, consultez la section conversions numériques implicites de l'article conversions numériques intégrées .

Pour les types référence, il existe toujours une conversion implicite entre une classe et l'une de ses interfaces ou classes de base directes ou indirectes. Aucune syntaxe spéciale n'est nécessaire, car une classe dérivée contient toujours tous les membres d'une classe de base.

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Conversions explicites

Toutefois, si une conversion ne peut pas être réalisée sans risque de perte d'informations, le compilateur exige une conversion explicite, aussi appelée *cast*. Un cast est un moyen d'informer explicitement le compilateur que vous avez l'intention d'effectuer la conversion et que vous savez que la perte de données peut se produire, ou le cast peut échouer au moment de l'exécution. Pour effectuer un cast, spécifiez le type voulu entre parenthèses devant la valeur ou la variable à convertir. Le programme suivant convertit un double en int. Le programme ne se compilera pas sans le cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Pour obtenir la liste complète des conversions numériques explicites prises en charge, consultez la section conversions numériques explicites de l'article conversions numériques intégrées .

Pour les types référence, un cast explicite est exigé si vous devez effectuer une conversion d'un type de base en type dérivé :

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

Une opération cast entre types référence ne change pas le type au moment de l'exécution de l'objet sous-jacent. Elle change seulement le type de la valeur utilisée comme référence à cet objet. Pour plus d'informations, consultez Polymorphisme.

Exceptions de conversion de type au moment de l'exécution

Dans certaines conversions de types référence, le compilateur ne peut pas déterminer si un cast sera valide. Il est possible qu'une opération cast dont la compilation fonctionne échoue au moment de l'exécution. Comme l'illustre l'exemple suivant, un cast de type qui échoue au moment de l'exécution provoque la levée d'une exception InvalidCastException.

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

La `Test` méthode a un `Animal` paramètre, par conséquent le cast explicite `a` de l'argument en un `Reptile` fait une supposition dangereuse. Il est plus sûr de ne pas faire d'hypothèses, mais de vérifier le type. C# fournit l'opérateur is pour vous permettre de tester la compatibilité avant d'effectuer réellement un cast. Pour plus d'informations, consultez Comment effectuer un cast en toute sécurité à l'aide de critères spéciaux et des opérateurs As et is.

spécification du langage C#

Pour plus d'informations, consultez la section Conversions de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Types
- Expression Cast
- Opérateurs de conversion définie par l'utilisateur
- Conversion de type généralisée
- Comment convertir une chaîne en nombre

Boxing et unboxing (Guide de programmation C#)

02/11/2020 • 8 minutes to read • Edit Online

Le boxing est la conversion d'un type valeur en type `object` ou en un type interface implémenté par ce type valeur. Lorsque le common language runtime (CLR) convertit un type valeur, il encapsule la valeur à l'intérieur d'une System.Object instance et la stocke sur le tas managé. L'unboxing extrait le type valeur de l'objet. La conversion boxing est implicite ; la conversion unboxing est explicite. Le concept de boxing et de unboxing repose sur la vue unifiée par C# du système de type, dans lequel une valeur de n'importe quel type peut être traitée en tant qu'objet.

Dans l'exemple suivant, la variable de type entier `i` est convertie (*boxed*) et assignée à l'objet `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

L'objet `o` peut ensuite être unboxed et assigné à la variable de type entier `i` :

```
o = 123;
i = (int)o;  // unboxing
```

Les exemples suivants montrent comment le boxing est utilisé dans C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
```

```
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Performances

Par rapport aux assignations simples, le boxing et l'unboxing sont des processus qui coûtent cher en calcul. Lorsqu'un type valeur est boxed, un nouvel objet doit être alloué et construit. À un degré moindre, le cast requis pour l'unboxing coûte également cher en calcul. Pour plus d'informations, consultez Performances.

Boxing

Le boxing est utilisé pour stocker des types valeur dans le tas rassemblé par garbage collection. Le boxing est une conversion implicite d'un type valeur en type `object` ou en un type interface implémenté par ce type valeur. Le boxing d'un type valeur alloue une instance d'objet sur le tas et copie la valeur dans le nouvel objet.

Dans l'exemple suivant, une variable de type valeur est déclarée :

```
int i = 123;
```

L'instruction ci-dessous réalise implicitement une opération de boxing sur la variable `i` :

```
// Boxing copies the value of i into object o.
object o = i;
```

Le résultat de cette instruction crée, sur la pile, un objet `o` qui fait référence à une valeur de type `int` sur le tas. Cette valeur est une copie de la valeur de type valeur qui a été assignée à la variable `i`. La différence entre les deux variables, `i` et `o`, est illustrée dans l'image de conversion boxing ci-dessous :

Il est également possible, mais jamais obligatoire, d'effectuer un boxing explicite comme dans l'exemple suivant :

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Description

Cet exemple utilise le boxing pour convertir une variable `i` (entier) en un objet `o`. Ensuite, la valeur `i` stockée dans la variable `123` est remplacée par la valeur `456`. L'exemple montre que le type valeur d'origine et que l'objet boxed utilisent des emplacements de mémoire distincts et peuvent, par conséquent, stocker des valeurs différentes.

Exemple

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Unboxing

L'unboxing est une conversion explicite du type `object` en un type valeur, ou d'un type interface en un type valeur qui implémente l'interface. Une opération d'unboxing comprend les étapes suivantes :

- Vérification de l'instance de l'objet pour s'assurer qu'il s'agit bien d'une valeur boxed du type valeur

spécifié.

- Copie de la valeur de l'instance dans la variable de type valeur.

Les instructions suivantes expliquent les opérations de boxing et d'unboxing :

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

L'illustration suivante montre le résultat de ces instructions :

Pour que l'unboxing de types valeur réussisse au moment de l'exécution, l'élément qui est unboxed doit être une référence à un objet précédemment créé par boxing d'une instance de ce type valeur. La tentative d'extraction de `null` provoque un NullReferenceException. La tentative d'extraction d'une référence vers un type de valeur incompatible provoque un InvalidCastException.

Exemple

L'exemple suivant montre un cas d'unboxing non valide et l' `InvalidCastException` qui en résulte. Avec `try` et `catch`, un message d'erreur est affiché lorsque l'erreur se produit.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

Sortie de ce programme :

```
Specified cast is not valid. Error: Incorrect unboxing.
```

Si vous modifiez l'instruction :

```
int j = (short) o;
```

to:

```
int j = (int) o;
```

la conversion sera réalisée, avec le résultat suivant :

`Unboxing OK.`

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Types référence
- Types de valeur

Comment convertir un tableau d'octets en int (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Cet exemple montre comment utiliser la classe BitConverter pour convertir un tableau d'octets en valeur int, puis la reconvertir en tableau d'octets. Vous devrez peut-être convertir des octets en un type de données intégré après avoir lu les octets sur le réseau. En plus de la méthode ToInt32 (Byte [ ] , Int32) dans l'exemple, le tableau suivant répertorie les méthodes de la BitConverter classe qui convertissent des octets (d'un tableau d'octets) en d'autres types intégrés.

TYPE RETOURNÉ	MÉTHODE
`bool`	ToBoolean (Byte [ ] , Int32)
`char`	ToChar (Byte [ ] , Int32)
`double`	ToDouble (Byte [ ] , Int32)
`short`	Toint16 ((Byte [ ] , Int32)
`int`	ToInt32 (Byte [ ] , Int32)
`long`	Toint64 ((Byte [ ] , Int32)
`float`	ToSingle (Byte [ ] , Int32)
`ushort`	Touint16 ((Byte [ ] , Int32)
`uint`	Touint32 ((Byte [ ] , Int32)
`ulong`	Touint64 ((Byte [ ] , Int32)

Exemple

Cet exemple Initialise un tableau d'octets, inverse le tableau si l'architecture de l'ordinateur est Little-endian (autrement dit, l'octet le moins significatif est stocké en premier), puis appelle la méthode ToInt32 (Byte [ ] , Int32) pour convertir quatre octets dans le tableau en un `int` . Le deuxième argument de ToInt32 (Byte [ ] , Int32) spécifie l'index de début du tableau d'octets.

> **NOTE**
>
> La sortie peut différer selon le endianness de l'architecture de votre ordinateur.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Exemple

Dans cet exemple, la méthode GetBytes(Int32) de la classe BitConverter est appelée pour convertir une valeur `int` en tableau d'octets.

> **NOTE**
>
> La sortie peut différer selon le endianness de l'architecture de votre ordinateur.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

Voir aussi

- BitConverter
- IsLittleEndian
- Types

Comment convertir une chaîne en nombre (Guide de programmation C#)

16/12/2020 • 7 minutes to read • Edit Online

Vous pouvez convertir une chaîne en nombre en appelant la `Parse` méthode ou `TryParse` trouvée sur les différents types numériques (`int` , `long` , `double` , etc.), ou en utilisant des méthodes de la System.Convert classe.

Il est légèrement plus efficace et simple d'appeler une `TryParse` méthode (par exemple, `int.TryParse("11", out number)`) ou une `Parse` méthode (par exemple, `var number = int.Parse("11")`). L'utilisation d'une méthode Convert s'avère plus utile pour les objets généraux qui implémentent IConvertible.

Vous utilisez `Parse` les `TryParse` méthodes ou sur le type numérique que la chaîne contient, comme le System.Int32 type. La méthode Convert.ToInt32 utilise Parse en interne. La `Parse` méthode retourne le nombre converti ; la `TryParse` méthode retourne une Boolean valeur qui indique si la conversion a réussi, et retourne le nombre converti dans un `out` paramètre. Si la chaîne n'est pas dans un format valide, `Parse` lève une exception, mais `TryParse` retourne `false` . Lorsque vous appelez une méthode `Parse` , vous devez toujours utiliser la gestion des exceptions pour intercepter une FormatException si l'opération d'analyse échoue.

Appel des méthodes Parse et TryParse

Les `Parse` `TryParse` méthodes et ignorent les espaces blancs au début et à la fin de la chaîne, mais tous les autres caractères doivent être des caractères qui forment le type numérique approprié (`int` ,,,,, etc `long` `ulong` `float` `decimal` .). La présence d'un espace blanc dans la chaîne qui forme le nombre génère une erreur. Par exemple, vous pouvez utiliser `decimal.TryParse` pour analyser « 10 », « 10,3 » ou « 10 », mais vous ne pouvez pas utiliser cette méthode pour analyser 10 à partir de « 10x », « 1 0 » (Notez l'espace incorporé), « 10.3 » (Notez l'espace incorporé), « 10E1 » (`float.TryParse` fonctionne ici), et ainsi de suite. Chaîne dont la valeur est `null` ou String.Empty ne parvient pas à être analysée correctement. Vous pouvez rechercher une chaîne Null ou vide avant de tenter de l'analyser en appelant la méthode String.IsNullOrEmpty.

L'exemple suivant illustre des appels à `Parse` et à `TryParse` qui ont réussi ou échoué.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(inputString);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

L'exemple suivant illustre une approche pour analyser une chaîne supposée inclure des caractères numériques de début (y compris des caractères hexadécimaux) et des caractères non numériques de fin. Il attribue dès le début des caractères valides à une nouvelle chaîne avant d'appeler la méthode TryParse. Étant donné que les chaînes à

analyser contiennent un petit nombre de caractères, l'exemple appelle la méthode String.Concat pour attribuer des caractères valides à une nouvelle chaîne. Pour une chaîne plus grande, la classe StringBuilder peut être utilisée à la place.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A' && char.ToUpperInvariant(c) <=
'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Appel des méthodes de conversion

Le tableau suivant répertorie quelques unes des méthodes de la classe Convert que vous pouvez utiliser pour convertir une chaîne en nombre.

TYPE NUMÉRIQUE	MÉTHODE
`decimal`	ToDecimal(String)

TYPE NUMÉRIQUE	MÉTHODE
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

L'exemple suivant appelle la Convert.ToInt32(String) méthode pour convertir une chaîne d'entrée en int. L'exemple intercepte les deux exceptions les plus courantes qui peuvent être levées par cette méthode, FormatException et OverflowException . Si le nombre obtenu peut être incrémenté sans dépasser Int32.MaxValue, l'exemple ajoute 1 au résultat et affiche la sortie.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
           }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

Voir aussi

- Types
- Comment déterminer si une chaîne représente une valeur numérique
- Exemple : utilitaire de mise en forme .NET Core WinForms (C#)

Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques (Guide de programmation C#)

16/12/2020 • 4 minutes to read • Edit Online

Ces exemples montrent comment effectuer les tâches suivantes :

- Obtenir la valeur hexadécimale de chaque caractère dans une chaîne (string).
- Obtenir la valeur char qui correspond à chaque valeur d'une chaîne hexadécimale.
- Convertir une `string` hexadécimale en int.
- Convertir une `string` hexadécimale en float.
- Convertir un tableau d'octets en `string` hexadécimale.

Exemple

Cet exemple génère la valeur hexadécimale de chaque caractère dans une `string`. Il convertit d'abord la `string` en un tableau de caractères. Ensuite, il appelle ToInt32(Char) sur chaque caractère afin d'obtenir sa valeur numérique. Pour finir, il représente le nombre sous forme hexadécimale dans une `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Exemple

Cet exemple analyse une `string` de valeurs hexadécimales et génère le caractère correspondant à chacune d'elles. Tout d'abord, elle appelle la méthode Split (char [ ]) pour obtenir chaque valeur hexadécimale en tant qu'individu `string` dans un tableau. Il appelle ensuite ToInt32(String, Int32) pour convertir la valeur hexadécimale en valeur décimale représentée sous la forme d'un entier. Il montre deux façons différentes d'obtenir le caractère

correspondant à ce code de caractère. La première technique utilise `string`, qui retourne le caractère correspondant à l'argument entier sous forme de ConvertFromUtf32(Int32). La deuxième technique effectue un cast explicite de l' `int` en char.

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Exemple

Cet exemple montre une autre façon de convertir un `string` hexadécimal en entier, en appelant la méthode Parse(String, NumberStyles).

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

Exemple

L'exemple suivant montre comment convertir un `string` hexadécimal en float à l'aide de la classe System.BitConverter et de la méthode UInt32.Parse.

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

Exemple

L'exemple suivant montre comment convertir un tableau d'octets en chaîne hexadécimale à l'aide de la classe System.BitConverter.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

Voir aussi

- Chaînes de format numériques standard
- Types
- Comment déterminer si une chaîne représente une valeur numérique

Utilisation du type dynamic (Guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

C# 4 introduit un nouveau type, `dynamic`. Il s'agit d'un type statique ; toutefois, un objet de type `dynamic` ignore la vérification des types statiques. Dans la plupart des cas, il fonctionne comme s'il était de type `object`. Au moment de la compilation, un élément de type `dynamic` est supposé prendre en charge n'importe quelle opération. Par conséquent, vous n'avez pas besoin de vous demander si l'objet obtient sa valeur d'une API COM, d'un langage dynamique tel qu'IronPython, du modèle DOM (Document Object Model) HTML, de la réflexion ou d'une autre partie du programme. Toutefois, si le code n'est pas valide, des erreurs sont détectées au moment de l'exécution.

Par exemple, si la méthode d'instance `exampleMethod1` du code suivant ne comporte qu'un seul paramètre, le compilateur reconnaît que le premier appel à la méthode, `ec.exampleMethod1(10, 4)`, n'est pas valide car il contient deux arguments. Cet appel entraîne une erreur du compilateur. Le deuxième appel à la méthode, `dynamic_ec.exampleMethod1(10, 4)`, n'est pas vérifié par le compilateur, car le type de `dynamic_ec` est `dynamic`. Par conséquent, aucune erreur de compilateur n'est signalée. Toutefois, l'erreur ne passe pas indéfiniment inaperçue. Elle est détectée au moment de l'exécution et provoque une exception runtime.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

Dans ces exemples, le rôle du compilateur consiste à regrouper les informations relatives à l'opération que chaque instruction propose d'exécuter sur l'objet ou l'expression de type `dynamic`. Au moment de l'exécution, les informations stockées sont examinées, et toute instruction non valide provoque une exception runtime.

Le résultat de la plupart des opérations dynamiques est lui-même de type `dynamic`. Par exemple, si vous placez le

pointeur de la souris sur `testSum` dans l'exemple suivant, IntelliSense affiche le type **(variable locale) dynamic testSum**.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Les opérations dans lesquelles le résultat n'est pas `dynamic` incluent :

- Les conversions de `dynamic` vers un autre type.
- Les appels de constructeur qui incluent des arguments de type `dynamic`.

Par exemple, le type de `testInstance` dans la déclaration suivante est `ExampleClass`, et non pas `dynamic` :

```
var testInstance = new ExampleClass(d);
```

Des exemples de conversion sont présentés dans la section suivante, "Conversions".

Conversions

Les conversions entre des objets dynamiques et d'autres types sont faciles à exécuter. Les développeurs peuvent ainsi passer du comportement dynamique au comportement non dynamique, et vice versa.

Tout objet peut être converti implicitement en type dynamique, comme illustré dans les exemples suivants.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Inversement, une conversion implicite peut être appliquée de façon dynamique à toute expression de type `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Résolution de surcharge avec des arguments de type dynamic

La résolution de surcharge a lieu au moment de l'exécution, et non pas au moment de la compilation, si un ou plusieurs arguments d'un appel de méthode sont de type `dynamic`, ou si le récepteur de l'appel de méthode est de type `dynamic`. Dans l'exemple suivant, si la seule méthode `exampleMethod2` accessible est définie pour accepter un argument de chaîne, l'envoi de `d1` en tant qu'argument n'entraîne pas d'erreur du compilateur, mais provoque une exception runtime. La résolution de surcharge échoue au moment de l'exécution car le type de `d1` est `int`, et `exampleMethod2` exige une chaîne.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic Language Runtime

Le Dynamic Language Runtime (DLR) est une API qui a été introduite dans .NET Framework 4. Il fournit l'infrastructure qui prend en charge le type `dynamic` en C#, ainsi que l'implémentation des langages de programmation dynamique tels qu'IronPython et IronRuby. Pour plus d'informations sur le DLR, consultez Vue d'ensemble du Dynamic Language Runtime.

COM interop

C# 4 inclut plusieurs fonctionnalités qui améliorent l'interopérabilité avec les API COM telles que les API Office Automation. L'utilisation du type `dynamic` et des arguments nommés et facultatifs fait partie des améliorations offertes.

De nombreuses méthodes COM autorisent une variation des types d'arguments et du type de retour en désignant les types comme `object`. Cela a nécessité un cast explicite des valeurs pour la coordination avec les variables fortement typées en C#. Si vous compilez à l'aide de l'option -Link (options du compilateur C#) , l'introduction du `dynamic` type vous permet de traiter les occurrences de `object` dans les signatures com comme si elles étaient de type `dynamic` , et donc d'éviter une grande partie du cast. Par exemple, les instructions suivantes permettent de comparer la façon dont vous accédez à une cellule d'une feuille de calcul Microsoft Office Excel avec le type `dynamic` et sans le type `dynamic`.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Rubriques connexes

TITRE	DESCRIPTION
dynamic	Décrit l'utilisation du mot clé `dynamic`.
Vue d'ensemble du Dynamic Language Runtime	Fournit une vue d'ensemble du DLR, qui est un environnement d'exécution ajoutant au Common Language Runtime (CLR) un ensemble de services pour les langages dynamiques.
Procédure pas à pas : création et utilisation d'objets dynamiques	Fournit des instructions pas à pas pour la création d'un objet dynamique personnalisé et pour la création d'un projet qui accède à une bibliothèque `IronPython`.

TITRE	DESCRIPTION
Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#	Montre comment créer un projet qui utilise les arguments nommés et facultatifs, le type `dynamic` et d'autres améliorations qui simplifient l'accès aux objets d'API Office.

Procédure pas à pas : Création et utilisation d'objets dynamiques (C# et Visual Basic)

16/12/2020 • 19 minutes to read • Edit Online

Les objets dynamiques exposent des membres tels que les propriétés et les méthodes au moment de l'exécution et non lors de la compilation. Cela vous permet de créer des objets utilisables avec des structures qui ne correspondent pas à un type ou un format statique. Par exemple, vous pouvez utiliser un objet dynamique pour référencer le modèle DOM (Document Object Model) HTML, qui peut contenir n'importe quelle combinaison d'attributs et d'éléments de balisage HTML valides. Étant donné que chaque document HTML est unique, les membres d'un document HTML particulier sont déterminés au moment de l'exécution. L'une des méthodes courantes pour référencer un attribut d'un élément HTML consiste à passer le nom de l'attribut à la méthode `GetProperty` de l'élément. Pour référencer l'attribut `id` de l'élément HTML `<div id="Div1">`, vous obtenez d'abord une référence à l'élément `<div>`, puis utilisez `divElement.GetProperty("id")`. Si vous utilisez un objet dynamique, vous pouvez référencer l'attribut `id` comme `divElement.id`.

Les objets dynamiques permettent également d'accéder facilement aux langages dynamiques comme IronPython et IronRuby. Vous pouvez utiliser un objet dynamique pour faire référence à un script dynamique qui est interprété au moment de l'exécution.

Vous référencez un objet dynamique à l'aide de la liaison tardive. En C#, vous spécifiez le type d'un objet à liaison tardive comme `dynamic`. En Visual Basic, vous spécifiez le type d'un objet à liaison tardive comme `Object`. Pour plus d'informations, consultez dynamic et Liaison anticipée et liaison tardive.

Vous pouvez créer des objets dynamiques personnalisés en utilisant les classes contenues dans l'espace de noms System.Dynamic. Par exemple, vous pouvez créer un ExpandoObject et spécifier les membres de cet objet au moment de l'exécution. Vous pouvez aussi créer votre propre type qui hérite de la classe DynamicObject. Vous pouvez ensuite substituer les membres de la classe DynamicObject pour fournir des fonctionnalités dynamiques au moment de l'exécution.

Dans cette procédure pas à pas, vous exécuterez les tâches suivantes :

- Créer un objet personnalisé qui expose dynamiquement le contenu d'un fichier texte comme propriétés d'un objet.
- Créer un projet qui utilise une bibliothèque `IronPython`.

Prérequis

Pour effectuer cette procédure pas à pas, vous devez utiliser IronPython pour .NET. Accédez à la page de téléchargement pour obtenir la version la plus récente.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Création d'un objet dynamique personnalisé

Le premier projet que vous créez dans cette procédure pas à pas définit un objet dynamique personnalisé qui

recherche dans le contenu d'un fichier texte. Le texte à rechercher est spécifié par le nom d'une propriété dynamique. Par exemple, si le code appelant spécifie `dynamicFile.Sample`, la classe dynamique retourne une liste générique de chaînes qui contient toutes les lignes du fichier qui commencent par « Sample ». La recherche respecte la casse. La classe dynamique prend également en charge deux arguments facultatifs. Le premier argument est une valeur enum de l'option de recherche qui spécifie que la classe dynamique doit rechercher des correspondances en début de ligne, en fin de ligne, ou n'importe où dans la ligne. Le deuxième argument spécifie que la classe dynamique doit supprimer des espaces à gauche et à droite dans chaque ligne avant d'effectuer la recherche. Par exemple, si le code appelant spécifie `dynamicFile.Sample(StringSearchOption.Contains)`, la classe dynamique recherche le terme « Sample » n'importe où dans une ligne. Si le code appelant spécifie `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, la classe dynamique recherche le terme « Sample » au début de chaque ligne et ne supprime ni les espaces à droite, ni les espaces à gauche. Le comportement par défaut de la classe dynamique consiste à rechercher une correspondance au début de chaque ligne et à supprimer les espaces à gauche et à droite.

Pour créer une classe dynamique personnalisée

1. Démarrez Visual Studio.

2. Dans le menu **fichier** , pointez sur **nouveau** , puis cliquez sur **projet**.

3. Dans la boîte de dialogue **Nouveau projet**, dans le volet **Types de projets**, vérifiez que **Windows** est sélectionné. Sélectionnez **Application console** dans le volet **Modèles**. Dans la zone **Nom**, tapez `DynamicSample`, puis cliquez sur **OK**. Le nouveau projet est créé.

4. Cliquez avec le bouton droit sur le projet DynamicSample, pointez sur **Ajouter**, puis cliquez sur **Classe**. Dans la zone **Nom**, tapez `ReadOnlyFile`, puis cliquez sur **OK**. Un nouveau fichier contenant la classe ReadOnlyFile est ajouté.

5. En haut du fichier ReadOnlyFile.cs ou ReadOnlyFile.vb, ajoutez le code suivant pour importer les espaces de noms System.IO et System.Dynamic.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

6. L'objet dynamique personnalisé utilise un enum pour déterminer les critères de recherche. Avant l'instruction de classe, ajoutez la définition d'enum suivante.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. Mettez à jour l'instruction de classe de façon à hériter de la classe `DynamicObject`, comme indiqué dans l'exemple de code suivant.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

8. Ajoutez le code suivant à la classe `ReadOnlyFile` afin de définir un champ privé pour le chemin du fichier et un constructeur pour la classe `ReadOnlyFile`.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

9. Ajoutez la méthode `GetPropertyValue` suivante à la classe `ReadOnlyFile`. La méthode `GetPropertyValue` prend les critères de recherche comme paramètre d'entrée et retourne les lignes d'un fichier texte qui correspondent aux critères de recherche. Les méthodes dynamiques fournies par la classe `ReadOnlyFile` appellent la méthode `GetPropertyValue` pour récupérer leurs résultats respectifs.

```
public List<string> GetPropertyValue(string propertyName,
                                     StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                     bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

10. Après la méthode `GetPropertyValue`, ajoutez le code suivant pour substituer la méthode TryGetMember de la classe DynamicObject. La méthode TryGetMember est appelée quand un membre d'une classe dynamique est demandé et qu'aucun argument n'est spécifié. L'argument `binder` contient les informations relatives au membre référencé et l'argument `result` référence le résultat retourné pour le membre spécifié. La méthode TryGetMember retourne une valeur booléenne qui retourne `true` si le membre demandé existe ; sinon, elle retourne la valeur `false`.

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

11. Après la méthode `TryGetMember`, ajoutez le code suivant pour substituer la méthode TryInvokeMember de la classe DynamicObject. La méthode TryInvokeMember est appelée quand un membre d'une classe dynamique est demandé avec des arguments. L'argument `binder` contient les informations relatives au membre référencé et l'argument `result` référence le résultat retourné pour le membre spécifié. L'argument `args` contient un tableau des arguments passés au membre. La méthode TryInvokeMember retourne une valeur booléenne qui retourne `true` si le membre demandé existe ; sinon, elle retourne la valeur `false`.

    La version personnalisée de la méthode `TryInvokeMember` s'attend à ce que le premier argument soit une valeur de l'enum `StringSearchOption` que vous avez défini dans une étape précédente. La méthode `TryInvokeMember` s'attend à ce que le deuxième argument soit une valeur booléenne. Si l'un des arguments ou les deux arguments sont des valeurs valides, ils sont passés à la méthode `GetPropertyValue` pour récupérer les résultats.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
                                          ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

12. Enregistrez et fermez le fichier.

Pour créer un exemple de fichier texte

1. Cliquez avec le bouton droit sur le projet DynamicSample, pointez sur **Ajouter**, puis cliquez sur **Nouvel élément**. Dans le volet **Modèles installés**, sélectionnez **Général**, puis sélectionnez le modèle **Fichier texte**. Laissez le nom par défaut TextFile1.txt dans la zone **Nom**, puis cliquez sur **Ajouter**. Un nouveau fichier texte est ajouté au projet.

2. Copiez le texte suivant dans le fichier TextFile1.txt.

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Enregistrez et fermez le fichier.

Pour créer un exemple d'application qui utilise l'objet dynamique personnalisé

1. Dans l'**Explorateur de solutions**, double-cliquez sur le fichier Module1.vb si vous utilisez Visual Basic ou sur le fichier Program.cs si vous utilisez Visual C#.

2. Ajoutez le code suivant à la procédure Main pour créer une instance de la classe `ReadOnlyFile` pour le fichier TextFile1.txt. Le code utilise la liaison tardive pour appeler des membres dynamiques et récupérer des lignes de texte qui contiennent la chaîne « Customer ».

```
dynamic rFile = new ReadOnlyFile(@"..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Enregistrez le fichier et appuyez sur CTRL+F5 pour générer et exécuter l'application.

Appel d'une bibliothèque dynamique de langage

Le projet suivant que vous créez dans cette procédure pas à pas permet d'accéder à une bibliothèque qui est écrite dans le langage dynamique IronPython.

Pour créer une classe dynamique personnalisée

1. Dans Visual Studio, dans le menu **fichier** , pointez sur **nouveau** , puis cliquez sur **projet**.

2. Dans la boîte de dialogue **Nouveau projet**, dans le volet **Types de projets**, vérifiez que **Windows** est sélectionné. Sélectionnez **Application console** dans le volet **Modèles**. Dans la zone **Nom**, tapez `DynamicIronPythonSample`, puis cliquez sur **OK**. Le nouveau projet est créé.

3. Si vous utilisez Visual Basic, cliquez avec le bouton droit sur le projet DynamicIronPythonSample, puis cliquez sur **Propriétés**. Cliquez sur l'onglet **références** . Cliquez sur le bouton **Ajouter** . Si vous utilisez Visual C#, dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le dossier **Références**, puis cliquez sur **Ajouter une référence**.

4. Sous l'onglet **Parcourir**, naviguez jusqu'au dossier dans lequel sont installées les bibliothèques IronPython. Par exemple, C:\Program Files\IronPython 2,6 pour .NET Framework 4,0. Sélectionnez les bibliothèques **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll** et **Microsoft.Dynamic.dll**. Cliquez sur **OK**.

5. Si vous utilisez Visual Basic, modifiez le fichier Module1.vb. Si vous utilisez Visual C#, modifiez le fichier Program.cs.

6. En haut du fichier, ajoutez le code suivant pour importer les espaces de noms `Microsoft.Scripting.Hosting` et `IronPython.Hosting` depuis les bibliothèques IronPython.

```
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
```

7. Dans la méthode Main, ajoutez le code suivant pour créer un objet `Microsoft.Scripting.Hosting.ScriptRuntime` pour héberger les bibliothèques IronPython. L'objet `ScriptRuntime` charge le module de bibliothèque IronPython random.py.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.6 for .NET 4.0\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
   My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
   "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

8. Une fois que le code a chargé le module random.py, ajoutez le code suivant pour créer un tableau d'entiers. Le tableau est passé à la méthode `shuffle` du module random.py, qui trie aléatoirement les valeurs dans le tableau.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("-------------------");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("-------------------")
Next
```

9. Enregistrez le fichier et appuyez sur CTRL+F5 pour générer et exécuter l'application.

Voir aussi

- System.Dynamic
- System.Dynamic.DynamicObject
- Utilisation du type dynamic
- Liaison précoce et tardive
- dynamic
- Implémenter des interfaces dynamiques (PDF téléchargeable sur Microsoft TechNet)

Classes et structs (Guide de programmation C#)

02/11/2020 • 12 minutes to read • Edit Online

Les classes et les structs sont deux des constructions de base du système de type commun dans .NET. Chacun est en substance une structure de données qui encapsule un ensemble de données et de comportements constituant une unité logique. Les données et comportements sont les *membres* de la classe ou du struct, et ils incluent ses méthodes, propriétés, événements, etc., comme indiqué plus loin dans cette rubrique.

Une déclaration de classe ou de struct est comme un plan utilisé pour créer des instances ou des objets au moment de l'exécution. Si vous définissez une classe ou un struct nommé `Person`, `Person` est le nom du type. Si vous déclarez et initialisez une variable `p` de type `Person`, `p` est dit objet ou instance de `Person`. Plusieurs instances du même type `Person` peuvent être créées, et chaque instance peut avoir des valeurs différentes dans ses propriétés et champs.

Une classe est un type de référence. Lorsqu'un objet de la classe est créé, la variable à laquelle l'objet est affecté conserve uniquement une référence à cette mémoire. Lorsque la référence d'objet est affectée à une variable, la nouvelle variable fait référence à l'objet d'origine. Les modifications apportées à une variable sont répercutées sur l'autre variable, car toutes deux font référence aux mêmes données.

Un struct est un type valeur. Lorsqu'un struct est créé, la variable à laquelle le struct est assigné contient les données réelles du struct. Lorsque le struct est affecté à une nouvelle variable, il est copié. La nouvelle variable et la variable d'origine contiennent par conséquent deux copies distinctes des mêmes données. Les modifications apportées à une copie n'affectent pas l'autre copie.

En général, les classes sont utilisées pour modéliser des comportements plus complexes, ou des données destinées à être modifiées après la création d'un objet de classe. Les structs conviennent mieux aux petites structures de données contenant principalement des données qui ne sont pas censées être modifiées après que la structure a été créé.

Pour plus d'informations, consultez classes, objetset types de structures.

Exemple

Dans l'exemple suivant, `CustomClass` dans l'espace de noms `ProgrammingGuide` a trois membres : un constructeur d'instance, une propriété nommée `Number` et une méthode nommée `Multiply`. La `Main` méthode de la `Program` classe crée une instance (objet) de `CustomClass` , et la méthode et la propriété de l'objet sont accessibles à l'aide de la notation par points.

```
using System;

namespace ProgrammingGuide
{
   // Class definition.
   public class CustomClass
   {
      // Class members.
      //
      // Property.
      public int Number { get; set; }

      // Method.
      public int Multiply(int num)
      {
          return num * Number;
      }

      // Instance Constructor.
      public CustomClass()
      {
          Number = 0;
      }
   }

   // Another class definition that contains Main, the program entry point.
   class Program
   {
      static void Main(string[] args)
      {
         // Create an object of type CustomClass.
         CustomClass custClass = new CustomClass();

         // Set the value of the public property.
         custClass.Number = 27;

         // Call the public method.
         int result = custClass.Multiply(4);
         Console.WriteLine($"The result is {result}.");
      }
   }
}
// The example displays the following output:
//      The result is 108.
```

Encapsulation

L'encapsulation est parfois considérée comme le premier pilier ou principe de la programmation orientée objet. D'après le principe d'encapsulation, une classe ou un struct peut spécifier le degré d'accessibilité de chacun de ses membres au code situé en dehors de la classe ou du struct. Les méthodes et variables qui ne sont pas destinées à être utilisées d'en dehors de la classe ou de l'assembly peuvent être masquées afin de limiter le risque d'erreurs ou de code malveillant exploitant une faille de sécurité.

Pour plus d'informations sur les classes, consultez les pages Classes et Objets.

Membres

Tous les champs, méthodes, constantes, propriétés et événements doivent être déclarés dans un type ; ils sont appelés *membres* du type. En C#, il n'existe aucune variable ou méthode globale, à la différence d'autres langages. Même le point d'entrée d'un programme, la méthode `Main`, doit être déclaré dans une classe ou un struct. La liste suivante inclut tous les types de membres qui peuvent être déclarés dans une classe ou un struct.

- Fields
- Constantes
- Propriétés
- Méthodes
- Constructeurs
- Événements
- Finaliseurs
- Indexeurs
- Opérateurs
- Types imbriqués

Accessibilité

Certaines méthodes et propriétés sont censées être appelées ou accessibles par le code qui se trouve à l'extérieur de votre classe ou de votre struct, connu sous le terme de *code client*. D'autres méthodes et propriétés peuvent être uniquement utilisables dans la classe ou le struct proprement dits. Il est important de limiter l'accessibilité de votre code afin que seul le code client prévu puisse y accéder. Vous pouvez spécifier l'accessibilité de vos types et de leurs membres vis-à-vis du code client à l'aide des modificateurs d'accès public, protected, internal, protected internal, private et private protected. L'accessibilité par défaut est `private`. Pour plus d'informations, consultez Modificateurs d'accès.

Héritage

Les classes (mais pas les structs) prennent en charge le concept d'héritage. Une classe qui dérive d'une autre classe (la *classe de base*) contient automatiquement tous les membres publics, protégés et internes de la classe de base, sauf ses constructeurs et finaliseurs. Pour plus d'informations, consultez les pages Héritage et Polymorphisme.

Les classes peuvent être déclarées comme abstraites, ce qui signifie qu'une ou plusieurs de leurs méthodes n'ont aucune implémentation. Bien que les classes abstraites ne puissent pas être instanciées directement, elles peuvent servir de classes de base à d'autres classes qui fournissent l'implémentation manquante. Les classes peuvent également être déclarées comme scellées pour empêcher d'autres classes d'hériter d'elles. Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Interfaces

Les classes et les structs peuvent hériter de plusieurs interfaces. Pour un type, hériter d'une interface signifie implémenter toutes les méthodes définies dans l'interface. Pour plus d'informations, consultez Interfaces.

Types génériques

Les classes et structs peuvent être définis avec un ou plusieurs paramètres de type. Le code client fournit le type lorsqu'il crée une instance du type. Par exemple, la classe List<T> de l'espace de noms System.Collections.Generic est définie avec un seul paramètre de type. Le code client crée une instance d'une `List<string>` ou d'une `List<int>` pour spécifier le type que contiendra la liste. Pour plus d'informations, consultez Génériques.

Types statiques

Les classes (mais pas les structs) peuvent être déclarées comme statiques. Une classe statique ne peut contenir que des membres statiques et ne peut pas être instanciée avec le mot clé new. Une copie de la classe est chargée en mémoire au chargement du programme, et ses membres sont accessibles par le biais du nom de la classe. Les classes et les structs peuvent contenir des membres statiques. Pour plus d'informations,

consultez la page Classes statiques et membres de classes statiques.

Types imbriqués

Une classe ou un struct peut être imbriqué dans une autre classe ou un autre struct. Pour plus d'informations, consultez types imbriqués.

Types partiels

Vous pouvez définir une partie d'une classe, d'un struct ou d'une méthode dans un fichier de code et une autre partie dans un fichier de code séparé. Pour plus d'informations, consultez la page Classes et méthodes partielles.

Initialiseurs d'objets

Vous pouvez instancier et initialiser des objets de classe et de struct, ainsi que des collections d'objets, sans appeler explicitement leur constructeur. Pour plus d'informations, consultez la page Initialiseurs d'objets et de collections.

Types anonymes

Dans les situations où il n'est pas pratique ou nécessaire de créer une classe nommée, par exemple pour remplir une liste avec des structures de données qu'il n'est pas nécessaire de conserver ou de transmettre à une autre méthode, on utilise des types anonymes. Pour plus d'informations, consultez types anonymes.

Méthodes d'extension

Vous pouvez « étendre » une classe sans créer de classe dérivée en créant un type séparé dont les méthodes peuvent être appelées comme si elles appartenaient au type d'origine. Pour plus d'informations, consultez Méthodes d'extension.

Variables locales implicitement typées

Dans une méthode de classe ou de struct, vous pouvez utiliser le type implicite pour indiquer au compilateur de déterminer le type approprié au moment de la compilation. Pour plus d'informations, consultez Variables locales implicitement typées.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#

Classes (Guide de programmation C#)

16/12/2020 • 9 minutes to read • Edit Online

Types référence

Un type défini comme class est un *type référence*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur Null tant que vous n'avez pas explicitement créé une instance de la classe à l'aide de l'opérateur new, ou que vous ne lui avez pas assigné un objet existant d'un type compatible, comme indiqué dans l'exemple suivant :

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

Quand l'objet est créé, une quantité de mémoire suffisante est allouée sur le tas managé de l'objet spécifié, et la variable contient uniquement une référence à l'emplacement de cet objet. Les types sur le tas managé entraînent une surcharge quand ils sont alloués et récupérés par la fonctionnalité de gestion automatique de la mémoire du CLR (appelée *garbage collection*). Toutefois, le garbage collection est également optimisé et, dans la plupart des cas, ne nuit pas aux performances. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire et garbage collection.

Déclaration de classes

Les classes sont déclarées à l'aide du mot clé class suivi d'un identificateur unique, comme l'illustre l'exemple suivant :

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

Le mot clé `class` est précédé du niveau d'accès. Comme public est utilisé dans ce cas, n'importe qui peut créer des instances de cette classe. Le nom de la classe suit le mot clé `class`. Le nom de la classe doit être un nom d'identificateur C# valide. Le reste de la définition est le corps de la classe, où le comportement et les données sont définis. Les champs, propriétés, méthodes et événements d'une classe sont désignés collectivement par le terme « *membres de classe* ».

Création d'objets

Bien qu'ils soient parfois employés indifféremment, une classe et un objet sont deux choses différentes. Une classe définit un type d'objet, mais il ne s'agit pas d'un objet en soi. Un objet, qui est une entité concrète basée sur une classe, est parfois désigné par le terme « instance de classe ».

Vous pouvez créer des objets en utilisant le mot clé new suivi du nom de la classe sur laquelle l'objet est basé, comme suit :

```
Customer object1 = new Customer();
```

Quand une instance d'une classe est créée, une référence à l'objet est repassée au programmeur. Dans l'exemple précédent, `object1` est une référence à un objet basé sur `Customer`. Cette référence fait référence au nouvel objet, mais elle ne contient pas ses données. En fait, vous pouvez créer une référence d'objet sans créer d'objet :

```
 Customer object2;
```

Nous vous déconseillons de créer des références d'objet comme celle-ci, sans référence à un objet, car toute tentative d'accès à un objet à l'aide d'une telle référence échoue au moment de l'exécution. Toutefois, une telle référence peut être faite pour faire référence à un objet, soit en créant un nouvel objet, soit en lui assignant un objet existant, tel que celui-ci :

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Ce code crée deux références d'objet qui font toutes deux référence au même objet. Toute modification apportée à l'objet par le biais de `object3` est donc reflétée dans les utilisations suivantes de `object4`. Les objets qui sont basés sur des classes étant désignés par référence, les classes sont appelées des « types référence ».

Héritage de classe

Les classes prennent entièrement en charge l' *héritage* , caractéristique fondamentale de la programmation orientée objet. Lorsque vous créez une classe, vous pouvez hériter de toute autre classe qui n'est pas définie comme sealed, et d'autres classes peuvent hériter de votre classe et remplacer les méthodes virtuelles de la classe. En outre, vous pouvez implémenter une ou plusieurs interfaces.

L'héritage se fait par le biais d'une *dérivation* , ce qui signifie qu'une classe est déclarée à l'aide d'une *classe de base* dont elle hérite les données et le comportement. Pour spécifier une classe de base, ajoutez deux-points et le nom de la classe de base après le nom de la classe dérivée, comme suit :

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Quand une classe déclare une classe de base, elle hérite de tous les membres de la classe de base à l'exception des constructeurs. Pour plus d'informations, consultez Héritage.

Contrairement à C++, une classe en C# ne peut hériter directement que d'une classe de base. Toutefois, une classe de base pouvant elle-même hériter d'une autre classe, une classe peut hériter indirectement de plusieurs classes de base. En outre, une classe peut implémenter directement une ou plusieurs interfaces. Pour plus d'informations, consultez Interfaces.

Une classe peut être déclarée abstract. Une classe abstraite contient des méthodes abstraites qui ont une définition de signature, mais aucune implémentation. Les classes abstraites ne peuvent pas être instanciées. Elles peuvent être utilisées uniquement à travers des classes dérivées qui implémentent les méthodes abstraites. En revanche, une classe sealed ne permet pas à d'autres classes de dériver d'elle. Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Les définitions de classe peuvent être fractionnées entre différents fichiers sources. Pour plus d'informations, consultez la page Classes et méthodes partielles.

Exemple

L'exemple suivant définit une classe publique qui contient une propriété implémentée automatiquement, une méthode et une méthode spéciale appelée un constructeur. Pour plus d'informations, consultez les rubriques Propriétés, Méthodes et Constructeurs. Les instances de la classe sont ensuite instanciées à l'aide du mot clé `new`.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Programmation orientée objet
- Polymorphisme
- Noms d'identificateurs
- Members (Membres)
- Méthodes
- Constructeurs
- Finaliseurs
- Objets

Objets (Guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

Une définition de classe ou de struct s'apparente à un plan qui spécifie ce que le type peut faire. Un objet est fondamentalement un bloc de mémoire qui a été alloué et configuré selon le plan. Un programme peut créer de nombreux objets de la même classe. Les objets sont également appelés instances. Ils peuvent être stockés dans une variable nommée, dans un tableau ou dans une collection. Le code client est le code qui utilise ces variables pour appeler les méthodes et accéder aux propriétés publiques de l'objet. Dans un langage orienté objet tel que C#, un programme classique se compose de plusieurs objets qui interagissent de façon dynamique.

> **NOTE**
>
> Les types statiques se comportent différemment de ce qui est décrit ici. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Instances struct et instances de classe

Étant donné que les classes sont des types référence, une variable d'un objet de classe conserve une référence à l'adresse de l'objet sur le tas managé. Si un deuxième objet du même type est assigné au premier objet, les deux variables font référence à l'objet à cette adresse. Ce point est abordé en détail plus loin dans cette rubrique.

Les instances de classes sont créées à l'aide de l'opérateur new. Dans l'exemple suivant, `Person` est le type, et `person1` et `person 2` sont des instances, ou objets, de ce type.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

Étant donné que les structs sont des types valeur, une variable d'un objet de struct conserve une copie de l'objet entier. Les instances de structs peuvent également être créées à l'aide de l'opérateur `new`, mais cela n'est pas obligatoire, comme illustré dans l'exemple suivant :

```
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

La mémoire pour `p1` et `p2` est allouée sur la pile de threads. Cette mémoire est récupérée avec le type ou la méthode où elle est déclarée. C'est l'une des raisons pour lesquelles les structs sont copiés au moment de l'assignation. En revanche, la mémoire allouée pour une instance de classe est récupérée automatiquement (garbage collection) par le common language runtime quand toutes les références à l'objet sont hors de la portée. Il n'est pas possible de détruire de façon déterministe un objet de classe comme vous pouvez le faire dans C++. Pour plus d'informations sur les garbage collection dans .NET, consultez garbage collection.

> **NOTE**
> L'allocation et la libération de mémoire sur le tas managé sont des opérations très optimisées dans le common language runtime. Dans la plupart des cas, il n'y a pas de différence significative sur le plan des performances entre l'allocation d'une instance de classe sur le tas et l'allocation d'une instance de struct sur la pile.

Identité de l'objet et égalité de la valeur

Quand vous effectuez une comparaison d'égalité entre deux objets, vous devez d'abord décider si vous souhaitez savoir si les deux variables représentent le même objet en mémoire, ou si les valeurs d'un ou de plusieurs de

leurs champs sont équivalentes. Si vous projetez de comparer les valeurs, vous devez établir si les objets sont des instances de types valeur (structs) ou de types référence (classes, délégués, tableaux).

- Pour déterminer si deux instances de classe référencent le même emplacement en mémoire (ce qui signifie qu'elles ont la même *identité*), utilisez la méthode statique Equals. (System.Object est la classe de base implicite pour tous les types valeur et types référence, y compris les classes et structs définis par l'utilisateur.)
- Pour déterminer si les champs d'instance dans deux instances de struct ont les mêmes valeurs, utilisez la méthode ValueType.Equals. Comme tous les structs héritent implicitement de System.ValueType, vous appelez directement la méthode sur votre objet, comme indiqué dans l'exemple suivant :

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

L'implémentation System.ValueType de `Equals` utilise la réflexion, car elle doit être en mesure de déterminer la nature des champs dans chaque struct. Quand vous créez vos propres structs, substituez la méthode `Equals` pour fournir un algorithme d'égalité efficace qui est propre à votre type.

- Pour déterminer si les valeurs des champs dans deux instances de classe sont égales, vous pouvez utiliser la méthode Equals ou l'opérateur ==. Toutefois, utilisez-les uniquement si la classe les a substitués ou surchargés pour fournir une définition personnalisée de ce que signifie « égalité » pour les objets de ce type. La classe peut également implémenter l'interface IEquatable<T> ou IEqualityComparer<T>. Les deux interfaces fournissent des méthodes qui peuvent être utilisées pour tester l'égalité des valeurs. Quand vous concevez vos propres classes qui substituent, veillez `Equals` à suivre les instructions indiquées dans comment définir l'égalité des valeurs pour un type et Object.Equals(Object) .

Sections connexes

Pour plus d'informations :

- Classes
- Constructeurs
- Finaliseurs
- Événements

Voir aussi

- Guide de programmation C#
- object
- Héritage
- class
- Types de structure
- nouvel opérateur
- Système de type commun

Héritage (Guide de programmation C#)

02/11/2020 • 11 minutes to read • Edit Online

L'héritage, avec l'encapsulation et le polymorphisme, est l'une des trois principales caractéristiques de la programmation orientée objet. L'héritage vous permet de créer de nouvelles classes qui réutilisent, étendent et modifient le comportement défini dans d'autres classes. La classe dont les membres sont hérités porte le nom de *classe de base* et la classe qui hérite de ces membres porte le nom de *classe dérivée*. Une classe dérivée ne peut avoir qu'une seule classe de base directe. Toutefois, l'héritage est transitif. Si `ClassC` est dérivée de, `ClassB` et `ClassB` est dérivée de `ClassA` , `ClassC` hérite des membres déclarés dans `ClassB` et `ClassA` .

> **NOTE**
>
> Les structs ne prennent pas en charge l'héritage, mais ils peuvent implémenter les interfaces. Pour plus d'informations, consultez Interfaces.

D'un point de vue conceptuel, une classe dérivée est une spécialisation de la classe de base. Par exemple, si vous avez une classe de base `Animal` , vous pouvez avoir une classe dérivée nommée `Mammal` et une autre classe dérivée nommée `Reptile` . Un `Mammal` est un `Animal` , et un `Reptile` est un `Animal` , mais chaque classe dérivée représente des spécialisations différentes de la classe de base.

Les déclarations d'interface peuvent définir une implémentation par défaut pour ses membres. Ces implémentations sont héritées par les interfaces dérivées et par les classes qui implémentent ces interfaces. Pour plus d'informations sur les méthodes d'interface par défaut, consultez l'article sur les interfaces dans la section Référence du langage.

Quand vous définissez une classe à dériver d'une autre classe, la classe dérivée obtient implicitement tous les membres de la classe de base, à l'exception de ses constructeurs et de ses finaliseurs. La classe dérivée réutilise le code de la classe de base sans avoir à la réimplémenter. Vous pouvez ajouter d'autres membres dans la classe dérivée. La classe dérivée étend les fonctionnalités de la classe de base.

L'illustration suivante montre une classe `WorkItem` qui représente un élément de travail dans un processus métier. Comme toutes les classes, elle dérive de System.Object et hérite de toutes ses méthodes. `WorkItem` ajoute cinq de ses propres membres. Ces membres incluent un constructeur, car les constructeurs ne sont pas hérités. La classe `ChangeRequest` hérite de `WorkItem` et représente un type particulier d'élément de travail. `ChangeRequest` ajoute deux membres supplémentaires aux membres qu'elle hérite de `WorkItem` et de Object. Elle doit ajouter son propre constructeur et ajoute également `originalItemID` . La propriété `originalItemID` permet à l'instance `ChangeRequest` d'être associée au `WorkItem` d'origine auquel s'applique la demande de modification.

L'exemple suivant montre comment les relations de classe de l'illustration précédente sont exprimées en langage C#. L'exemple montre également comment `WorkItem` substitue la méthode virtuelle Object.ToString, et comment la classe `ChangeRequest` hérite de l'implémentation `WorkItem` de la méthode. Le premier bloc définit les classes :

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
```

```
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

Le bloc suivant montre comment utiliser les classes de base et dérivées :

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Méthodes abstraites et virtuelles

Lorsqu'une classe de base déclare une méthode en tant que `virtual`, une classe dérivée peut `override` la méthode avec sa propre implémentation. Si une classe de base déclare un membre en tant que `abstract`, cette méthode doit être substituée dans toute classe non abstraite qui hérite directement de cette classe. Si une classe dérivée est abstraite, elle hérite des membres abstraits sans les implémenter. Les membres virtuels et abstraits sont la base du polymorphisme, qui est la deuxième caractéristique principale de la programmation orientée objet. Pour plus d'informations, consultez Polymorphisme.

Classes de base abstraites

Pour empêcher l'instanciation directe, vous pouvez déclarer une classe comme étant abstract à l'aide de l'opérateur new. Une classe abstraite ne peut être utilisée que si une nouvelle classe en est dérivée. Une classe abstraite peut contenir une ou plusieurs signatures de méthode qui sont également déclarées comme abstraites. Ces signatures spécifient les paramètres et la valeur de retour, mais n'ont aucune implémentation (corps de méthode). Une classe abstraite ne doit pas contenir de membres abstraits ; Toutefois, si une classe contient un membre abstrait, la classe elle-même doit être déclarée comme abstraite. Les classes dérivées qui ne sont pas abstraites elles-mêmes doivent fournir l'implémentation pour toutes les méthodes abstraites d'une classe de base abstraite. Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Interfaces

Une *interface* est un type référence qui définit un jeu de membres. Toutes les classes et les structs qui implémentent cette interface doivent implémenter ce jeu de membres. Une interface peut définir une implémentation par défaut pour tout ou partie de ces membres. Une classe peut implémenter plusieurs interfaces, même si elle ne peut dériver que d'une seule classe de base directe.

Les interfaces sont utilisées pour définir des fonctionnalités spécifiques pour les classes qui n'ont pas nécessairement une relation « est une ». Par exemple, l' System.IEquatable<T> interface peut être

implémentée par n'importe quelle classe ou structure pour déterminer si deux objets du type sont équivalents (Toutefois, le type définit l'équivalence). IEquatable<T> n'implique pas le même type de relation « est un » qui existe entre une classe de base et une classe dérivée (par exemple, un `Mammal` est un `Animal`). Pour plus d'informations, consultez Interfaces.

Prévention des dérivations ultérieures

Une classe peut empêcher d'autres classes d'hériter d'elle, ou de l'un de ses membres, en déclarant elle-même ou le membre comme `sealed` . Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Masquage de classe dérivée des membres de classe de base

Une classe dérivée peut masquer des membres de la classe de base en déclarant les membres à l'aide du même nom et de la même signature. Le `new` modificateur peut être utilisé pour indiquer explicitement que le membre n'est pas destiné à être une substitution du membre de base. L'utilisation de `new` n'est pas obligatoire, mais un avertissement du compilateur est généré si `new` n'est pas utilisé. Pour plus d'informations, consultez Gestion de version avec les mots clés override et new et Savoir quand utiliser les mots clés override et new.

Voir aussi

- Guide de programmation C#
- Classes et structs
- class

Polymorphisme (Guide de programmation C#)

02/11/2020 • 14 minutes to read • Edit Online

Le polymorphisme est souvent considéré comme le troisième pilier d'une programmation orientée objet, après l'encapsulation et l'héritage. Le polymorphisme est le mot grec qui signifie « plusieurs formes » et il prend deux aspects distincts :

- Au moment de l'exécution, les objets d'une classe dérivée peuvent être traités comme des objets d'une classe de base dans les paramètres de méthode et les collections ou les tableaux. Lorsque ce polymorphisme se produit, le type déclaré de l'objet n'est plus identique à son type au moment de l'exécution.
- Les classes de base peuvent définir et implémenter des *méthodes* virtuelles , et les classes dérivées peuvent les substituer , ce qui signifie qu'elles fournissent leur propre définition et implémentation. Au moment de l'exécution, quand le code client appelle la méthode, le CLR recherche le type au moment de l'exécution et appelle cette substitution de la méthode virtuelle. Dans votre code source, vous pouvez appeler une méthode sur une classe de base et provoquer l'exécution de la version d'une classe dérivée de la méthode.

Les méthodes virtuelles vous permettent d'utiliser des groupes d'objets liés de façon uniforme. Par exemple, si vous avez une application de dessin qui permet à un utilisateur de créer différents types de formes sur une surface de dessin. Vous ne savez pas au moment de la compilation les types de formes spécifiques que l'utilisateur va créer. Cependant, l'application doit conserver une trace des différents types de formes créés et les mettre à jour en réponse aux actions de la souris. Vous pouvez utiliser le polymorphisme pour résoudre ce problème en deux étapes :

1. Créer une hiérarchie de classe dans laquelle chaque classe de forme dérive d'une classe de base commune.
2. Utiliser une méthode virtuelle pour appeler la méthode appropriée dans une classe dérivée via un seul appel à la méthode de classe de base.

Tout d'abord, créez une classe de base appelée `Shape`, et des classes dérivées telles que `Rectangle`, `Circle`, et `Triangle`. Donnez à la classe `Shape` une méthode virtuelle appelée `Draw`, et substituez-la dans chaque classe dérivée pour dessiner une forme spécifique représentée par la classe. Créez un `List<Shape>` objet et ajoutez `Circle`, `Triangle` et `Rectangle` à celui-ci.

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

Pour mettre à jour la surface de dessin, utilisez une boucle foreach pour l'itération dans la liste et appelez la méthode `Draw` sur chaque objet `Shape` de la liste. Bien que chaque objet de la liste ait un type déclaré `Shape`, il s'agit du type au moment de l'exécution (la version substituée de la méthode dans chaque classe dérivée) qui sera appelée.

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used whereever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

En C#, chaque type est polymorphique, car tous les types, y compris les types définis par l'utilisateur, héritent de Object.

Vue d'ensemble du polymorphisme

Membres virtuels

Lorsqu'une classe dérivée hérite d'une classe de base, elle obtient toutes les méthodes, les champs, les propriétés et les événements de la classe de base. Le concepteur de la classe dérivée a des choix différents pour le comportement des méthodes virtuelles :

- La classe dérivée peut substituer des membres virtuels dans la classe de base, définissant un nouveau comportement.
- La classe dérivée hérite de la méthode de classe de base la plus proche sans la substituer, en préservant le comportement existant, mais en permettant à d'autres classes dérivées de substituer la méthode.
- La classe dérivée peut définir une nouvelle implémentation non virtuelle de ces membres qui masquent les implémentations de la classe de base.

Une classe dérivée ne peut substituer un membre de classe de base que si le membre de classe de base est déclaré comme étant virtual ou abstract. Le membre dérivé doit utiliser le mot clé override pour indiquer explicitement que la méthode est conçue pour participer à l'appel virtuel. Le code suivant est fourni à titre d'exemple :

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Les champs ne peuvent pas être virtuels ; seules les méthodes, les propriétés, les événements et les indexeurs peuvent être virtuels. Quand une classe dérivée est substituée à un membre virtuel, ce membre est appelé lors de l'accès à une instance de cette classe en tant qu'instance de la classe de base. Le code suivant est fourni à titre d'exemple :

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Les méthodes et propriétés virtuelles permettent aux classes dérivées d'étendre une classe de base sans avoir besoin d'utiliser l'implémentation de classe de base d'une méthode. Pour plus d'informations, consultez Versioning avec les mots clés override et new. Une interface fournit un autre moyen pour définir une méthode ou un ensemble de méthodes dont l'implémentation est confiée aux classes dérivées. Pour plus d'informations, consultez Interfaces.

Masquer les membres de la classe de base avec les nouveaux membres

Si vous souhaitez que votre classe dérivée ait un membre portant le même nom qu'un membre dans une classe de base, vous pouvez utiliser le mot clé New pour masquer le membre de la classe de base. Le mot clé `new` est placé avec le type de retour d'un membre de classe qui est remplacé. Le code suivant est fourni à titre d'exemple :

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Les membres masqués de la classe de base sont accessibles à partir du code client en effectuant un cast de l'instance de la classe dérivée vers une instance de la classe de base. Par exemple :

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Empêcher les classes dérivées de substituer des membres virtuels

Les membres virtuels restent virtuels, quel que soit le nombre de classes déclarées entre le membre virtuel et la classe qui l'a déclaré à l'origine. Si la classe `A` déclare un membre virtuel, et que `B` la classe dérive de `A`, et que `C` la classe dérive de `B`, `C` la classe hérite du membre virtuel et peut le substituer, que la classe ait déclaré ou non `B` une substitution pour ce membre. Le code suivant est fourni à titre d'exemple :

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

Une classe dérivée peut arrêter l'héritage virtuel en déclarant une substitution comme étant sealed. L'arrêt de l'héritage nécessite de placer le `sealed` mot clé avant le `override` mot clé dans la déclaration de membre de classe. Le code suivant est fourni à titre d'exemple :

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

Dans l'exemple précédent, la méthode `DoWork` n'est plus virtuelle à une classe dérivée de `C`. Il est toujours virtuel pour les instances de `C`, même s'ils sont convertis en type `B` ou en type `A`. Les méthodes sealed peuvent être remplacées par des classes dérivées à l'aide du `new` mot clé, comme le montre l'exemple suivant :

```
public class D : C
{
    public new void DoWork() { }
}
```

Dans ce cas, si `DoWork` est appelé sur `D` à l'aide d'une variable de type `D`, le nouveau `DoWork` est appelé. Si une variable de type `C`, `B` ou `A` est utilisée pour accéder à une instance de `D`, un appel à `DoWork` suit les règles d'héritage virtuel, en routant ces appels à l'implémentation de `DoWork` sur la classe `C`.

Accéder aux membres virtuels de la classe de base à partir de classes dérivées

Une classe dérivée qui a remplacé ou écrasé une méthode ou une propriété peut encore accéder à la méthode ou à la propriété dans la classe de base avec le mot clé `base`. Le code suivant est fourni à titre d'exemple :

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

Pour plus d'informations, consultez base.

> **NOTE**
>
> Il est recommandé que les membres virtuels utilisent `base` pour appeler l'implémentation de classe de base de ce membre dans leur propre implémentation. L'exécution du comportement de classe de base permet à la classe dérivée de se concentrer sur l'implémentation du comportement spécifique à la classe dérivée. Si l'implémentation de classe de base n'est pas appelée, la classe dérivée doit rendre son comportement compatible avec le comportement de la classe de base.

Contenu de cette section

- Versioning avec les mots clés override et new
- Savoir quand utiliser les mots clés override et new
- Comment : substituer la méthode ToString

Voir aussi

- Guide de programmation C#
- Héritage
- Classes abstract et sealed et membres de classe
- Méthodes
- Événements
- Propriétés
- Indexeurs
- Types

Versioning avec les mots clés override et new (Guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Le langage C# est conçu de telle sorte que la gestion de version entre les classes de base et les classes dérivées dans différentes bibliothèques puisse évoluer et préserver une compatibilité descendante. Cela signifie, par exemple, que l'introduction dans une classe de base d'un nouveau membre portant le même nom qu'un membre dans une classe dérivée est totalement prise en charge par C# et n'engendre pas de comportement imprévisible. Cela signifie également qu'une classe doit indiquer de façon explicite si une méthode est conçue pour se substituer à une méthode héritée ou s'il s'agit d'une nouvelle méthode qui masque une méthode héritée portant un nom similaire.

Dans C#, les classes dérivées peuvent contenir des méthodes portant le même nom que des méthodes de classe de base.

- Si la méthode dans la classe dérivée n'est pas précédée des mots clés new ou override, le compilateur émet un avertissement et la méthode se comporte comme si le mot clé `new` était présent.
- Si la méthode dans la classe dérivée est précédée du mot clé `new`, la méthode est définie comme étant indépendante de la méthode dans la classe de base.
- Si la méthode dans la classe dérivée est précédée du mot clé `override`, les objets de la classe dérivée appelleront cette méthode plutôt que la méthode de la classe de base.
- Afin d'appliquer le `override` mot clé à la méthode dans la classe dérivée, la méthode de classe de base doit être définie comme étant virtuelle.
- La méthode de la classe de base peut être appelée à partir de la classe dérivée à l'aide du mot clé `base`.
- Les mots clés `override`, `virtual` et `new` peuvent également être appliqués aux propriétés, indexeurs et événements.

Par défaut, les méthodes C# ne sont pas virtuelles. Si une méthode est déclarée comme virtuelle, toute classe héritant de la méthode peut implémenter sa propre version. Pour créer une méthode virtuelle, le modificateur `virtual` est utilisé dans la déclaration de méthode de la classe de base. La classe dérivée peut ensuite remplacer la méthode virtuelle de base avec le mot clé `override` ou masquer la méthode virtuelle dans la classe de base avec le mot clé `new`. Si ni le mot clé `override` ni le mot clé `new` ne sont spécifiés, le compilateur émettra un avertissement et la méthode dans la classe dérivée masquera la méthode dans la classe de base.

Pour illustrer ceci dans un exemple pratique, supposons un instant que la Société A ait créé une classe nommée `GraphicsClass` que votre programme utilise. Les éléments suivants représentent `GraphicsClass` :

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Votre société utilise cette classe, et vous l'utilisez pour dériver votre propre classe, en ajoutant une nouvelle méthode :

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Votre application est utilisée sans problèmes, jusqu'à ce que la Société A publie une nouvelle version de `GraphicsClass`, qui ressemble au code suivant :

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

La nouvelle version de `GraphicsClass` contient maintenant une méthode nommée `DrawRectangle`. Initialement, rien ne se produit. La nouvelle version est encore compatible au format binaire avec l'ancienne version. Tous les logiciels que vous avez déployés continueront de fonctionner, même si la nouvelle classe est installée sur ces systèmes informatiques. Tout appel existant à la méthode `DrawRectangle` continuera de faire référence à votre version, dans votre classe dérivée.

Toutefois, dès que vous recompilerez votre application à l'aide de la nouvelle version de `GraphicsClass`, vous recevrez un avertissement du compilateur, CS0108. Cet avertissement vous informe que vous devez envisager la façon dont vous souhaitez que votre méthode `DrawRectangle` se comporte dans votre application.

Si vous souhaitez que votre méthode remplace la nouvelle méthode de la classe de base, utilisez le mot clé `override` :

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

Le mot clé `override` garantit que tous les objets dérivés de `YourDerivedGraphicsClass` utiliseront la version de classe dérivée de `DrawRectangle`. Les objets dérivés de `YourDerivedGraphicsClass` peuvent toujours accéder à la version de la classe de base de `DrawRectangle` à l'aide du mot clé base :

```
base.DrawRectangle();
```

Si vous ne souhaitez pas que votre méthode se substitue à la nouvelle méthode de la classe de base, prenez en compte les considérations suivantes. Pour éviter toute confusion entre les deux méthodes, vous pouvez renommer votre méthode. Ceci peut prendre du temps et constituer un risque d'erreur ou bien ne pas être tout simplement pratique dans certains cas. Toutefois, si votre projet est relativement petit, vous pouvez utiliser les options de refactorisation de Visual Studio pour renommer la méthode. Pour plus d'informations, consultez Refactorisation des classes et des types (Concepteur de classes).

Sinon, vous pouvez éviter de recevoir l'avertissement à l'aide du mot clé `new` dans la définition de votre classe dérivée :

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

L'utilisation du mot clé `new` indique au compilateur que votre définition masque la définition qui est contenue dans la classe de base. Il s'agit du comportement par défaut.

Substitution et sélection de méthode

Lorsqu'une méthode est appelée sur une classe, le compilateur C# sélectionne la meilleure méthode à appeler si plusieurs méthodes sont compatibles avec l'appel, comme lorsque deux méthodes portent les mêmes noms et dont les paramètres sont compatibles avec les paramètres transmis. Les méthodes suivantes seraient compatibles :

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

Quand `DoWork` est appelée sur une instance de `Derived`, le compilateur C# essaie d'abord de rendre l'appel compatible avec les versions de `DoWork` déclarées à l'origine sur `Derived`. Les méthodes override ne sont pas considérées comme déclarées sur une classe ; il s'agit de nouvelles implémentations d'une méthode déclarée sur une classe de base. Le compilateur C# essaiera de faire correspondre l'appel à une méthode substituée portant le même nom et avec des paramètres compatibles uniquement s'il ne parvient pas à faire correspondre l'appel de méthode à une méthode d'origine sur `Derived`. Par exemple :

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Étant donné que la variable `val` peut être convertie implicitement en un double, le compilateur C# appelle `DoWork(double)` au lieu de `DoWork(int)`. Il existe deux façons d'éviter ceci. Premièrement, évitez de déclarer de nouvelles méthodes portant le même nom que des méthodes virtuelles. Deuxièmement, vous pouvez indiquer au compilateur C# d'appeler la méthode virtuelle en le faisant chercher dans la liste de méthodes de la classe de base par casting de l'instance `Derived` en `Base`. Étant donné que la méthode est virtuelle, l'implémentation de `DoWork(int)` sur `Derived` sera appelée. Par exemple :

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Pour obtenir d'autres exemples des mots clés `new` et `override`, consultez Savoir quand utiliser les mots clés override et new.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Méthodes
- Héritage

Savoir quand utiliser les mots clés override et new (Guide de programmation C#)

02/11/2020 • 15 minutes to read • Edit Online

En C#, une méthode dans une classe dérivée peut avoir le même nom qu'une méthode dans la classe de base. Vous pouvez spécifier le mode d'interaction des méthodes avec les mots clés new et override. Le modificateur `override` *étend* la méthode `virtual` de la classe de base, tandis que le modificateur `new` *masque* une méthode de classe de base accessible. La différence est illustrée dans les exemples de cette rubrique.

Dans une application console, déclarez les deux classes suivantes, `BaseClass` et `DerivedClass`. `DerivedClass` hérite de `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

Dans la méthode `Main`, déclarez les variables `bc`, `dc` et `bcdc`.

- `bc` est de type `BaseClass` et sa valeur est de type `BaseClass`.
- `dc` est de type `DerivedClass` et sa valeur est de type `DerivedClass`.
- `bcdc` est de type `BaseClass` et sa valeur est de type `DerivedClass`. Il s'agit de la variable à surveiller.

Comme `bc` et `bcdc` ont le type `BaseClass`, ils ne peuvent accéder directement qu'à `Method1`, sauf si vous effectuez un cast. La variable `dc` peut accéder à `Method1` et `Method2`. Ces relations sont illustrées dans le code suivant.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Ensuite, ajoutez la méthode `Method2` suivante à `BaseClass`. La signature de cette méthode correspond à celle de la méthode `Method2` dans `DerivedClass`.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Comme `BaseClass` a maintenant une méthode `Method2`, une deuxième instruction d'appel peut être ajoutée pour les variables `BaseClass``bc` et `bcdc`, comme illustré dans le code suivant.

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

Quand vous générez le projet, vous constatez que l'ajout de la méthode `Method2` dans `BaseClass` génère un avertissement. L'avertissement indique que la méthode `Method2` dans `DerivedClass` masque la méthode `Method2` dans `BaseClass`. Il est recommandé d'utiliser le mot clé `new` dans la définition de `Method2` si vous avez l'intention de provoquer ce résultat. Sinon, vous pouvez renommer l'une des méthodes `Method2` pour résoudre l'avertissement, mais ce n'est pas toujours pratique.

Avant d'ajouter `new`, exécutez le programme pour afficher la sortie produite par les instructions d'appel supplémentaires. Les résultats suivants s'affichent.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

Le mot clé `new` conserve les relations qui génèrent cette sortie, mais supprime l'avertissement. Les variables de type `BaseClass` continuent d'accéder aux membres de `BaseClass` et la variable de type `DerivedClass` continue

d'accéder aux membres dans `DerivedClass` avant de prendre en compte les membres hérités de `BaseClass`.

Pour supprimer l'avertissement, ajoutez le modificateur `new` à la définition de `Method2` dans `DerivedClass`, comme illustré dans le code suivant. Le modificateur peut être ajouté avant ou après `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Réexécutez le programme pour vérifier que la sortie n'a pas changé. Vérifiez aussi que l'avertissement ne s'affiche plus. À l'aide de `new`, vous déclarez que vous êtes conscient que le membre qu'il modifie masque un membre hérité de la classe de base. Pour plus d'informations sur le masquage de nom par héritage, consultez new, modificateur.

Pour comparer ce comportement aux effets de l'utilisation de `override`, ajoutez la méthode suivante à `DerivedClass`. Le modificateur `override` peut être ajouté avant ou après `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Ajoutez le modificateur `virtual` à la définition de `Method1` dans `BaseClass`. Le modificateur `virtual` peut être ajouté avant ou après `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Réexécutez le projet. Notez en particulier les deux dernières lignes de la sortie suivante.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

L'utilisation du modificateur `override` permet à `bcdc` d'accéder à la méthode `Method1` définie dans `DerivedClass`. En règle générale, il s'agit du comportement souhaité dans les hiérarchies d'héritage. Vous voulez que les objets dont les valeurs sont créées à partir de la classe dérivée utilisent les méthodes définies dans la classe dérivée. Vous obtenez ce comportement en utilisant `override` pour étendre la méthode de classe de base.

Le code suivant contient l'exemple complet.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

L'exemple suivant illustre un comportement similaire dans un autre contexte. L'exemple définit trois classes : une classe de base nommée `Car` et deux de ses classes dérivées, `ConvertibleCar` et `Minivan`. La classe de base contient une méthode `DescribeCar`. La méthode affiche une description de base d'une voiture et appelle ensuite `ShowDetails` pour fournir des informations supplémentaires. Chacune des trois classes définit une méthode `ShowDetails`. Le modificateur `new` est utilisé pour définir `ShowDetails` dans la classe `ConvertibleCar`. Le modificateur `override` est utilisé pour définir `ShowDetails` dans la classe `Minivan`.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

L'exemple vérifie la version de `ShowDetails` qui est appelée. La méthode suivante, `TestCars1`, déclare une instance de chaque classe, puis appelle `DescribeCar` sur chaque instance.

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` génère la sortie suivante. Notez en particulier les résultats de `car2`, qui ne correspondent probablement pas à ce que vous attendiez. Le type de l'objet est `ConvertibleCar`, mais `DescribeCar` n'accède pas à la version de `ShowDetails` qui est défini dans la classe `ConvertibleCar`, car cette méthode est déclarée avec le modificateur `new`, et non `override`. Par conséquent, un objet `ConvertibleCar` affiche la même description qu'un objet `Car`. Comparez les résultats pour `car3`, qui est un objet `Minivan`. Dans ce cas, la méthode `ShowDetails` déclarée dans la classe `Minivan` substitue la méthode `ShowDetails` déclarée dans la classe `Car` et la description affichée décrit un minivan.

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` crée une liste d'objets de type `Car`. Les valeurs des objets sont instanciées à partir des classes `Car`, `ConvertibleCar` et `Minivan`. `DescribeCar` est appelé sur chaque élément de la liste. Le code suivant illustre la définition de `TestCars2`.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

La sortie suivante s'affiche. Notez qu'elle est identique à la sortie affichée par `TestCars1`. La méthode `ShowDetails` de la classe `ConvertibleCar` n'est pas appelée, que le type de l'objet soit `ConvertibleCar`, comme dans `TestCars1`, ou `Car`, comme dans `TestCars2`. À l'inverse, `car3` appelle la méthode `ShowDetails` à partir de la classe `Minivan` dans les deux cas, que son type soit `Minivan` ou `Car`.

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Les méthodes `TestCars3` et `TestCars4` terminent l'exemple. Ces méthodes appellent `ShowDetails` directement, d'abord à partir des objets déclarés avec le type `ConvertibleCar` et `Minivan` (`TestCars3`), puis à partir des objets déclarés avec le type `Car` (`TestCars4`). Le code suivant définit ces deux méthodes.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Les méthodes produisent la sortie suivante, qui correspond aux résultats du premier exemple de cette rubrique.

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

Le code suivant illustre le projet complet et sa sortie.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
```

```
namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
```

```
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
```

```
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

Voir aussi

- Guide de programmation C#
- Classes et structs
- Versioning avec les mots clés override et new
- base
- abstraction

Comment substituer la méthode ToString (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

En C#, chaque classe ou struct hérite implicitement de la classe Object. Ainsi, chaque objet en C# obtient la méthode ToString, qui retourne une représentation sous forme de chaîne de cet objet. Par exemple, toutes les variables de type `int` ont une méthode `ToString`, ce qui leur permet de retourner leur contenu sous forme de chaîne :

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

Quand vous créez une classe ou un struct personnalisé, vous devez substituer la méthode ToString pour fournir des informations sur votre type au code client.

Pour plus d'informations sur l'utilisation de chaînes de format et d'autres types de mise en forme personnalisée avec la méthode `ToString`, consultez Mise en forme des types.

> **IMPORTANT**
>
> Quand vous choisissez les informations à fournir par l'intermédiaire de cette méthode, déterminez si votre classe ou struct sera utilisé par du code non fiable. Veillez à ne pas fournir d'informations susceptibles d'être exploitées par du code malveillant.

Pour substituer la méthode `ToString` dans votre classe ou struct :

1. Déclarez une méthode `ToString` avec les modificateurs et le type de retour suivants :

   ```
   public override string ToString(){}
   ```

2. Implémentez la méthode pour qu'elle retourne une chaîne.

   L'exemple suivant retourne le nom de la classe en plus des données propres à une instance particulière de la classe.

   ```
   class Person
   {
       public string Name { get; set; }
       public int Age { get; set; }

       public override string ToString()
       {
           return "Person: " + Name + " " + Age;
       }
   }
   ```

   Vous pouvez tester la méthode `ToString` comme illustré dans l'exemple de code suivant :

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

Voir aussi

- IFormattable
- Guide de programmation C#
- Classes et structs
- Chaînes
- string
- override
- virtuels
- Mise en forme des types

Membres (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Les classes et les structs ont des membres qui représentent leurs données et leur comportement. Les membres d'une classe incluent tous les membres déclarés de la classe, ainsi que tous les membres (à l'exception des constructeurs et des finaliseurs) déclarés dans toutes les classes de sa hiérarchie d'héritage. Les membres privés dans les classes de base sont hérités, mais ne sont pas accessibles à partir des classes dérivées.

Le tableau suivant liste les types de membres qu'une classe ou un struct peut contenir :

MEMBRE	DESCRIPTION
Fields	Les champs sont des variables déclarées au niveau de la classe. Un champ peut être un type numérique intégré ou une instance d'une autre classe. Par exemple, une classe de calendrier peut avoir un champ qui contient la date actuelle.
Constantes	Les constantes sont des champs dont la valeur est définie au moment de la compilation et ne peut pas être modifiée.
Propriétés	Les propriétés sont des méthodes sur une classe, accessibles comme si elles étaient des champs de cette classe. Une propriété peut fournir une protection pour un champ de classe afin d'éviter qu'il ne soit modifié sans que de l'objet en ait connaissance.
Méthodes	Les méthodes définissent les actions qu'une classe peut effectuer. Les méthodes peuvent accepter des paramètres qui fournissent des données d'entrée et peuvent retourner des données de sortie au moyen de paramètres. Les méthodes peuvent également retourner une valeur directement, sans utiliser de paramètre.
Événements	Les événements fournissent des notifications d'occurrences, comme des clics de bouton ou l'exécution réussie d'une méthode, sur d'autres objets. Les événements sont définis et déclenchés à l'aide de délégués.
Opérateurs	Les opérateurs surchargés sont considérés comme des membres de type. Quand vous surchargez un opérateur, vous le définissez sous forme de méthode statique publique dans un type. Pour plus d'informations, consultez Surcharge d'opérateur.
Indexeurs	Les indexeurs permettent à un objet d'être indexé d'une manière similaire aux tableaux.
Constructeurs	Les constructeurs sont des méthodes qui sont appelées quand l'objet est créé. Ils sont souvent utilisés pour initialiser les données d'un objet.

MEMBRE	DESCRIPTION
Finaliseurs	Les finaliseurs sont très rarement utilisés en C#. Ce sont des méthodes appelées par le moteur d'exécution quand l'objet est sur le point d'être supprimé de la mémoire. Ils sont généralement utilisés pour vérifier que toutes les ressources qui doivent être libérées sont gérées correctement.
Types imbriqués	Les types imbriqués sont des types déclarés dans un autre type. Les types imbriqués sont souvent utilisés pour décrire les objets utilisés uniquement par les types qui les contiennent.

Voir aussi

- Guide de programmation C#
- Catégories

Classes abstract et sealed et membres de classe (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Le mot clé abstract vous permet de créer des classes et des membres de class qui sont incomplets et doivent être implémentés dans une classe dérivée.

Le mot clé sealed vous permet d'empêcher l'héritage d'une classe ou de certains membres de classe qui étaient précédemment marqués comme virtual.

Classes abstraites et membres de classe

Les classes peuvent être déclarées comme abstraites en plaçant le mot clé `abstract` avant la définition de classe. Par exemple :

```
public abstract class A
{
    // Class members here.
}
```

Une classe abstraite ne peut pas être instanciée. Le but d'une classe abstraite est de fournir une définition commune d'une classe de base pouvant être partagée par plusieurs classes dérivées. Par exemple, une bibliothèque de classes peut définir une classe abstraite qui est utilisée comme un paramètre pour beaucoup de ses fonctions, et exiger des programmeurs qui l'utilisent de fournir leur propre implémentation de la classe en créant une classe dérivée.

Les classes abstraites peuvent également définir des méthodes abstraites. Pour ce faire, ajoutez le mot clé `abstract` avant le type de retour de la méthode. Par exemple :

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Les méthodes abstraites n'ont pas d'implémentation. Ainsi, la définition de la méthode est suivie d'un point-virgule au lieu d'un bloc de méthode normal. Les classes dérivées de la classe abstraite doivent implémenter toutes les méthodes abstraites. Lorsqu'une classe abstraite hérite d'une méthode virtuelle d'une classe de base, la classe abstraite peut substituer la méthode virtuelle par une méthode abstraite. Par exemple :

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Si une méthode `virtual` est déclarée `abstract`, elle demeure virtuelle pour toutes les classes qui héritent de la classe abstraite. Une classe qui hérite d'une méthode abstraite ne peut pas accéder à l'implémentation d'origine de la méthode. Dans l'exemple précédent, `DoWork` sur la classe F ne peut pas appeler `DoWork` sur la classe D. De cette manière, une classe abstraite peut forcer les classes dérivées à fournir de nouvelles implémentations de méthode pour les méthodes virtuelles.

Classes sealed et membres de classe

Les classes peuvent être déclarées comme sealed en plaçant le mot clé `sealed` avant la définition de classe. Par exemple :

```
public sealed class D
{
    // Class members here.
}
```

Une classe sealed ne peut pas être utilisée comme une classe de base. Pour cette raison, elle ne peut pas également être une classe abstraite. Les classes sealed empêchent la dérivation. Étant donné qu'elles ne peuvent pas être utilisées comme une classe de base, quelques optimisations au moment de l'exécution permettent d'accélérer un peu les appels aux membres des classes sealed.

Une méthode, un indexeur, une propriété ou un événement sur une classe dérivée qui substitue un membre virtuel de la classe de base peut déclarer ce membre comme sealed. Cela nie l'aspect virtuel du membre pour toute classe dérivée ultérieure. Pour ce faire, mettez le mot clé `sealed` avant le mot clé override dans la déclaration de membre de classe. Par exemple :

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

Voir aussi

- Guide de programmation C#
- Classes et structs

- Héritage
- Méthodes
- Fields
- Comment : définir des propriétés abstraites

Classes statiques et membres de classe statique (Guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Une classe statique est fondamentalement identique à une classe non statique, à une différence près : une classe statique ne peut pas être instanciée. En d'autres termes, vous ne pouvez pas utiliser l'opérateur new pour créer une variable du type classe. Étant donné qu'il n'y a aucune variable d'instance, vous accédez aux membres d'une classe statique en utilisant le nom de classe lui-même. Par exemple, si vous avez une classe statique nommée `UtilityClass` qui a une méthode statique publique nommée `MethodA`, vous appelez la méthode comme illustré dans l'exemple suivant :

```
UtilityClass.MethodA();
```

Une classe statique peut servir de conteneur pratique pour des ensembles de méthodes qui opèrent simplement sur des paramètres d'entrée et n'ont pas à obtenir ou définir de champs d'instance internes. Par exemple, dans la bibliothèque de classes .NET, la System.Math classe statique contient des méthodes qui effectuent des opérations mathématiques, sans qu'il soit nécessaire de stocker ou de récupérer des données qui sont uniques à une instance particulière de la Math classe. Autrement dit, vous appliquez les membres de la classe en spécifiant le nom de la classe et le nom de la méthode, comme illustré dans l'exemple suivant.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Comme c'est le cas avec tous les types de classe, les informations de type pour une classe statique sont chargées par le Runtime .NET lorsque le programme qui référence la classe est chargé. Le programme ne peut pas spécifier exactement quand la classe est chargée. Toutefois, il est garanti qu'elle sera chargée, que ses champs seront initialisés et que son constructeur statique sera appelé avant que la classe soit référencée pour la première fois dans votre programme. Un constructeur statique est appelé une seule fois et une classe statique reste en mémoire pendant la durée de vie du domaine d'application dans lequel votre programme réside.

> **NOTE**
>
> Pour créer une classe non statique permettant la création d'une seule instance d'elle-même, consultez Implémentation d'un singleton en C#.

La liste suivante fournit les fonctionnalités principales d'une classe statique :

- Elle contient uniquement des membres statiques.
- Elle ne peut pas être instanciée.
- Elle est verrouillée (sealed).

- Elle ne peut pas contenir de constructeurs d'instances.

La création d'une classe statique est par conséquent très semblable à la création d'une classe contenant uniquement des membres statiques et un constructeur privé. Un constructeur privé empêche la classe d'être instanciée. L'avantage de l'utilisation d'une classe statique est que le compilateur peut vérifier qu'aucun membre d'instance n'a été ajouté par erreur. Le compilateur garantit que les instances de cette classe ne peuvent pas être créées.

Les classes statiques sont scellées (sealed) et ne peuvent par conséquent pas être héritées. Elles ne peuvent hériter d'aucune classe à part Object. Les classes static ne peuvent pas contenir un constructeur d'instance. Toutefois, ils peuvent contenir un constructeur statique. Les classes non statiques doivent également définir un constructeur statique si la classe contient des membres statiques qui requièrent une initialisation non triviale. Pour plus d'informations, consultez Constructeurs statiques.

Exemple

Voici un exemple d'une classe statique qui contient deux méthodes qui convertissent la température des degrés Celsius en degrés Fahrenheit et des degrés Fahrenheit en degrés Celsius :

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;
```

```
            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Membres static

Une classe non statique peut contenir des méthodes, des champs, des propriétés ou des événements statiques. Le membre statique peut être appelé sur une classe même quand aucune instance de la classe n'a été créée. Le membre statique est toujours accessible par le nom de la classe, et non par le nom de l'instance. Une seule copie d'un membre statique existe, quel que soit le nombre d'instances de la classe qui ont été créées. Les méthodes et les propriétés statiques ne peuvent pas accéder à des champs et des événements non statiques dans leur type conteneur, et elles ne peuvent pas accéder à une variable d'instance de n'importe quel objet, sauf si elle est explicitement passée dans un paramètre de méthode.

Il est plus courant de déclarer une classe non statique avec certains membres statiques que de déclarer une classe entière comme statique. Deux utilisations courantes des champs statiques consistent à conserver un décompte du nombre d'objets qui ont été instanciés ou à stocker une valeur qui doit être partagée entre toutes les instances.

Les méthodes statiques peuvent être surchargées mais pas substituées, car elles appartiennent à la classe et non pas à une instance de la classe.

Bien qu'un champ ne puisse pas être déclaré en tant que `static const`, un champ const est essentiellement statique dans son comportement. Il appartient au type, pas aux instances du type. Par conséquent, `const` les champs sont accessibles à l'aide de la même `ClassName.MemberName` notation que celle utilisée pour les champs statiques. Aucune instance d'objet n'est requise.

C# ne prend pas en charge les variables locales statiques (autrement dit, les variables déclarées dans la portée de la méthode).

Vous déclarez des membres de classe statique en utilisant le mot clé `static` avant le type de retour du membre, comme illustré dans l'exemple suivant :

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

Les membres statiques sont initialisés avant le premier accès au membre statique et avant que le constructeur statique, s'il en existe un, soit appelé. Pour accéder à un membre de classe statique, utilisez le nom de la classe au lieu d'un nom de variable pour spécifier l'emplacement du membre, comme illustré dans l'exemple suivant :

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Si votre classe contient des champs statiques, fournissez un constructeur statique qui les initialise quand la classe est chargée.

Un appel à une méthode statique génère une instruction d'appel en langage MSIL (Microsoft Intermediate Language), tandis qu'un appel à une méthode d'instance génère une `callvirt` instruction, qui vérifie également les références d'objet null. Toutefois, la plupart du temps, l'écart de performances entre les deux n'est pas significatif.

Spécification du langage C#

Pour plus d'informations, voir Classes statiques et Membres statiques et d'instances dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- static
- Classes
- class
- Constructeurs statiques
- Constructeurs d'instances

Modificateurs d'accès (Guide de programmation C#)

02/11/2020 • 8 minutes to read • Edit Online

Tous les types et membres de type ont un niveau d'accessibilité. Le niveau d'accessibilité détermine s'ils peuvent être utilisés à partir d'un autre code de votre assembly ou d'autres assemblys. Utilisez les modificateurs d'accès suivants pour spécifier l'accessibilité d'un type ou d'un membre lorsque vous le déclarez :

- public: le type ou le membre est accessible par tout autre code du même assembly ou d'un autre assembly qui le référence.
- privé: le type ou le membre est accessible uniquement par le code dans le même `class` ou `struct` .
- protégé: il est possible d'accéder au type ou au membre uniquement par du code dans le même `class` ou dans un `class` qui est dérivé de celui-ci `class` .
- interne: le type ou le membre est accessible par n'importe quel code dans le même assembly, mais pas à partir d'un autre assembly.
- protected internal: le type ou le membre est accessible par n'importe quel code de l'assembly dans lequel il est déclaré, ou à partir d'un dérivé `class` dans un autre assembly.
- protégé privé: le type ou le membre est accessible uniquement dans son assembly déclarant, par le code dans le même `class` ou dans un type dérivé de celui-ci `class` .

Les exemples suivants montrent comment spécifier des modificateurs d'accès sur un type et un membre :

```
public class Bicycle
{
    public void Pedal() { }
}
```

Tous les modificateurs d'accès ne sont pas valides pour tous les types ou membres dans tous les contextes. Dans certains cas, l'accessibilité d'un membre de type est restreinte par l'accessibilité de son type conteneur.

Accessibilité des classes et des structs

Les classes et les structs déclarés directement dans un espace de noms (en d'autres termes, qui ne sont pas imbriqués dans d'autres classes ou structs) peuvent être `public` ou `internal` . `Internal` est la valeur par défaut si aucun modificateur d'accès n'est spécifié.

Les membres de struct, y compris les classes et structs imbriqués, peuvent être déclarés `public` , `internal` ou `private` . Les membres de classe, y compris les classes et structs imbriqués, peuvent être `public` , `protected internal` , `protected` ,, `internal` `private protected` ou `private` . Les membres de classe et de struct, y compris les classes et structs imbriqués, ont `private` accès par défaut. Les types imbriqués privés ne sont pas accessibles à partir de l'extérieur du type conteneur.

Les classes dérivées ne peuvent pas avoir une accessibilité supérieure à leurs types de base. Vous ne pouvez pas déclarer une classe publique `B` qui dérive d'une classe interne `A` . Si cela est autorisé, cela aurait pour effet de rendre `A` public, car tous les `protected` `internal` membres ou de `A` sont accessibles à partir de la classe dérivée.

Vous pouvez autoriser d'autres assemblys spécifiques à accéder à vos types internes à l'aide de

`InternalsVisibleToAttribute` . Pour plus d'informations, consultez assemblys friend.

Accessibilité des membres de classe et de struct

Vous pouvez déclarer les membres de classe (notamment les classes et structs imbriqués) avec l'un des six types d'accès. Les membres de struct ne peuvent pas être déclarés en tant que `protected` , `protected internal` ou, car les `private protected` structs ne prennent pas en charge l'héritage.

Normalement, l'accessibilité d'un membre n'est pas supérieure à l'accessibilité du type qui le contient. Toutefois, un `public` membre d'une classe interne peut être accessible depuis l'extérieur de l'assembly si le membre implémente des méthodes d'interface ou remplace des méthodes virtuelles définies dans une classe de base publique.

Le type d'un champ, d'une propriété ou d'un événement de membre doit être au moins aussi accessible que le membre lui-même. De même, le type de retour et les types de paramètre d'une méthode, d'un indexeur ou d'un délégué doivent être au moins aussi accessibles que le membre lui-même. Par exemple, vous ne pouvez pas avoir de `public` méthode `M` qui retourne une classe `C` à moins que `C` ne soit également `public` . De même, vous ne pouvez pas avoir une `protected` propriété de type `A` si `A` est déclaré comme `private` .

Les opérateurs définis par l'utilisateur doivent toujours être déclarés comme `public` et `static` . Pour plus d'informations, consultez Surcharge d'opérateur.

Les finaliseurs ne peuvent pas avoir de modificateurs d'accessibilité.

Pour définir le niveau d'accès pour `class` un `struct` membre ou, ajoutez le mot clé approprié à la déclaration de membre, comme indiqué dans l'exemple suivant.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

Autres types

Les interfaces déclarées directement dans un espace de noms peuvent être `public` ou `internal` et, tout comme les classes et les structs, les interfaces ont par défaut `internal` accès. Les membres d'interface sont `public` par défaut, car l'objectif d'une interface est de permettre à d'autres types d'accéder à une classe ou un struct. Les déclarations de membres d'interface peuvent inclure n'importe quel modificateur d'accès. Cela est très utile pour les méthodes statiques afin de fournir les implémentations courantes nécessaires à tous les implémenteurs d'une classe.

Les membres de l'énumération sont toujours `public` , et aucun modificateur d'accès ne peut être appliqué.

Les délégués se comportent comme des classes et des structs. Par défaut, ils ont `internal` accès lorsqu'ils sont déclarés directement dans un espace de noms, et sont `private` accessibles lorsqu'ils sont imbriqués.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Interfaces
- priv
- public
- intérieurs
- protected
- protected internal
- protégé privé
- class
- modélis
- interface

Champs (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Un *champ* est une variable de tout type qui est déclarée directement dans une classe ou un struct. Les champs sont *membres* de leur type contenant.

Une classe ou un struct peut avoir des champs d'instance, des champs statiques, ou les deux. Les champs d'instance sont spécifiques à une instance de type. Si vous avez une classe T avec un champ d'instance F, vous pouvez créer deux objets de type T et modifier la valeur de F dans chaque objet, sans affecter la valeur de l'autre objet. En revanche, un champ statique appartient à la classe, et il est partagé entre toutes les instances de cette classe. Vous pouvez accéder au champ statique uniquement à l'aide du nom de la classe. Si vous accédez au champ static par un nom d'instance, vous recevez une erreur de compilation CS0176 .

En règle générale, vous devez utiliser des champs uniquement pour les variables dont l'accessibilité est privée ou protégée. Les données que votre classe expose au code client doivent être fournies par le biais de méthodes, de Propriétéset d' indexeurs. L'utilisation de ces constructions pour l'accès indirect aux champs internes permet d'éviter les valeurs d'entrée non valides. Un champ privé qui stocke les données exposées par une propriété publique est appelé *magasin de stockage* ou *champ de stockage*.

En général, les champs stockent les données qui doivent être accessibles à plusieurs méthodes de classe et doivent être stockées plus longtemps que la durée de vie d'une méthode. Par exemple, une classe qui représente une date de calendrier peut avoir trois champs de type entier : un pour le mois, un pour le jour et un pour l'année. Les variables qui ne sont pas utilisées en dehors de la portée d'une méthode doivent être déclarées comme des *variables locales* dans le corps de la méthode.

Les champs sont déclarés dans le bloc Class en spécifiant le niveau d'accès du champ, suivi du type du champ, puis du nom du champ. Par exemple :

```
public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    public DateTime Date
    {
        get
        {
            return date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    // Public method also exposes date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt != null && dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}
```

Pour accéder à un champ dans un objet, ajoutez un point après le nom de l'objet, suivi du nom du champ, comme dans `objectname.fieldname`. Par exemple :

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

Une valeur initiale peut être attribuée à un champ à l'aide de l'opérateur d'assignation, lorsque le champ est déclaré. Pour assigner automatiquement le champ `day` à `"Monday"`, par exemple, vous devez déclarer `day`, comme dans l'exemple suivant :

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

Les champs sont initialisés juste avant le constructeur pour l'instance d'objet qui est appelée. Si le constructeur assigne la valeur d'un champ, la valeur assignée pendant la déclaration du champ sera remplacée. Pour plus d'informations, consultez Utilisation de constructeurs.

> **NOTE**
>
> Un initialiseur de champ ne peut pas référencer d'autres champs d'instance.

Les champs peuvent être marqués comme public, Private, protected, Internal, protected internalou Private protected. Ces modificateurs d'accès définissent comment les utilisateurs de la classe peuvent accéder aux champs. Pour plus d'informations, consultez Modificateurs d'accès.

Facultatif : vous pouvez déclarer un champ comme static. Ainsi, le champ est disponible à tout moment pour les appelants, même si aucune instance de la classe n'existe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Un champ peut être déclaré comme readonly. Un champ en lecture seule ne peut recevoir de valeur que pendant l'initialisation ou dans un constructeur. Un champ `static readonly` est très similaire à une constante, à ceci près que le compilateur C# n'a pas accès à la valeur d'un champ statique en lecture seule au moment de la compilation, mais seulement au moment de l'exécution. Pour plus d'informations, consultez Constantes.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Utilisation de constructeurs
- Héritage
- Modificateurs d'accès
- Classes abstract et sealed et membres de classe

Constantes (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Les constantes sont des valeurs immuables qui sont connues au moment de la compilation et qui ne changent pas pendant la durée de vie du programme. Les constantes sont déclarées avec le modificateur const. Seuls les types intégrés C# (à l'exception de System.Object) peuvent être déclarés comme `const` . Les types définis par l'utilisateur, notamment les classes, les structs et les tableaux, ne peuvent pas être `const`. Utilisez le modificateur readonly pour créer une classe, un struct ou un tableau initialisé une fois au moment de l'exécution (par exemple, dans un constructeur) et qui ne peut pas être modifié ultérieurement.

C# ne prend pas en charge les propriétés, les événements ou les méthodes `const`.

Le type enum vous permet de définir des constantes nommées pour les types intégrés intégraux (par exemple, `int`, `uint`, `long`, etc.). Pour plus d'informations, consultez enum.

Les constantes doivent être initialisées à mesure qu'elles sont déclarées. Par exemple :

```
class Calendar1
{
    public const int Months = 12;
}
```

Dans cet exemple, la constante `Months` est toujours 12, et ne peut pas être modifiée, même par la classe elle-même. En fait, quand le compilateur rencontre un identificateur constant dans du code source C# (par exemple, `Months`), il substitue la valeur littérale directement dans le code en langage intermédiaire (IL, Intermediate Language) qu'il produit. Étant donné qu'aucune adresse de variable n'est associée à une constante au moment de l'exécution, les champs `const` ne peuvent pas être passés par référence et ne peuvent pas apparaître comme l-value dans une expression.

> **NOTE**
>
> Procédez avec précaution lorsque vous faites référence à des valeurs constantes définies dans un autre code telles que des DLL. Si une nouvelle version de la DLL définit une nouvelle valeur pour la constante, votre programme conserve l'ancienne valeur littérale jusqu'à ce qu'elle soit recompilée sur la nouvelle version.

Plusieurs constantes du même type peuvent être déclarées en même temps, par exemple :

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

L'expression utilisée pour initialiser une constante peut référencer une autre constante si elle ne crée pas de référence circulaire. Par exemple :

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Les constantes peuvent être marquées comme public, private, protected, internal, protected internal ou private protected. Ces modificateurs d'accès définissent la façon dont les utilisateurs de la classe peuvent accéder à la constante. Pour plus d'informations, consultez Modificateurs d'accès.

Les constantes sont accessibles comme si elles étaient des champs static, car la valeur de la constante est identique pour toutes les instances du type. Vous n'utilisez pas le mot clé `static` pour les déclarer. Les expressions ne se trouvant pas dans la classe qui définit la constante doivent utiliser le nom de la classe, un point et le nom de la constante pour accéder à la constante. Par exemple :

```
int birthstones = Calendar.Months;
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Propriétés
- Types
- seulement
- Immutability in C# Part One: Kinds of Immutability

Comment définir des propriétés abstraites (Guide de programmation C#)

16/12/2020 • 3 minutes to read • Edit Online

L'exemple suivant montre comment définir des propriétés abstract. Une déclaration de propriété abstraite ne fournit pas une implémentation des accesseurs de propriété ; elle déclare que la classe prend en charge des propriétés, mais laisse l'implémentation de l'accesseur aux classes dérivées. L'exemple suivant montre comment implémenter les propriétés abstraites héritées d'une classe de base.

Cet exemple se compose de trois fichiers, chacun étant compilé individuellement, et l'assembly obtenu est référencé par la compilation suivante :

- abstractshape.cs : classe `Shape` qui contient une propriété `Area` abstraite.
- shapes.cs : sous-classes de la classe `Shape`.
- shapetest.cs : programme de test destiné à afficher les zones de certains objets dérivés de `Shape`.

Pour compiler l'exemple, utilisez la commande suivante :

```
csc abstractshape.cs shapes.cs shapetest.cs
```

Cette commande crée le fichier exécutable shapetest.exe.

Exemple

Ce fichier déclare la classe `Shape` qui contient la propriété `Area` du type `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Les modificateurs sur la propriété sont placés sur la déclaration de propriété proprement dite. Par exemple :

  ```
  public abstract double Area
  ```

- Quand vous déclarez une propriété abstraite (telle que `Area` dans cet exemple), vous indiquez simplement quels les accesseurs de propriété sont disponibles, mais vous ne les implémentez pas. Dans cet exemple, seul un accesseur get est disponible, donc la propriété est en lecture seule.

Exemple

Le code suivant présente trois sous-classes de `Shape` et indique comment elles substituent la propriété `Area` pour fournir leur propre implémentation.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

Exemple

Le code suivant montre un programme de test qui crée un certain nombre d'objets dérivés de `Shape` et imprime les zones correspondantes.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

Voir aussi

- Guide de programmation C#
- Classes et structs
- Classes abstract et sealed et membres de classe
- Propriétés

Comment définir des constantes en C#

16/12/2020 • 2 minutes to read • Edit Online

Les constantes sont des champs dont les valeurs sont définies au moment de la compilation et ne sont pas modifiables. Utilisez des constantes pour fournir des noms explicites au lieu de littéraux numériques (« nombres magiques ») pour les valeurs spéciales.

> **NOTE**
>
> En C#, la directive de préprocesseur #define ne peut pas être utilisée pour définir des constantes de la manière généralement utilisée en C et C++.

Pour définir des valeurs constantes de types intégraux (`int`, `byte`, etc.), utilisez un type énuméré. Pour plus d'informations, consultez enum.

Pour définir des constantes non intégrales, une approche consiste à les regrouper dans une classe statique unique nommée `Constants`. Cette opération exige que toutes les références aux constantes soient précédées par le nom de classe, comme dans l'exemple suivant.

Exemple

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}
class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
    }
}
```

L'utilisation du qualificateur de nom de classe permet de garantir que les personnes qui utilisent la constante, dont vous-même, comprennent qu'il s'agit d'une valeur constante et qu'elle ne peut pas être modifiée.

Voir aussi

- Classes et structs

Propriétés (Guide de programmation C#)

02/11/2020 • 8 minutes to read • Edit Online

Une propriété est un membre qui fournit un mécanisme flexible pour la lecture, l'écriture ou le calcul de la valeur d'un champ privé. Les propriétés peuvent être utilisées comme s'il s'agissait de membres de données publics, mais ce sont en fait des méthodes spéciales appelées *accesseurs*. Elles permettent aux données d'être facilement accessibles tout en soutenant quand même la sécurité et la flexibilité des méthodes.

Vue d'ensemble des propriétés

- Les propriétés permettent à une classe d'exposer un moyen public d'obtenir et de définir des valeurs, tout en masquant le code d'implémentation ou de vérification.
- Un accesseur de propriété get est utilisé pour retourner la valeur de la propriété et un accesseur de propriété set est utilisé pour affecter une nouvelle valeur. Ces accesseurs peuvent avoir différents niveaux d'accès. Pour plus d'informations, consultez Restriction d'accessibilité de l'accesseur.
- Le mot clé value est utilisé pour définir la valeur assignée par l'accesseur `set`.
- Les propriétés peuvent être *en lecture-écriture* (elles ont un accesseur `get` et un accesseur `set`), *en lecture seule* (elles ont un accesseur `get`, mais pas d'accesseur `set`) ou *en écriture seule* (elles ont un accesseur `set`, mais pas d'accesseur `get`). Les propriétés en écriture seule sont rares et sont généralement utilisées pour restreindre l'accès aux données sensibles.
- Les propriétés simples qui ne nécessitent pas de code d'accesseur personnalisé peuvent être implémentées en tant que définitions de corps d'expression ou en tant que propriétés implémentées automatiquement.

Propriétés avec des champs de stockage

Un modèle de base pour l'implémentation d'une propriété consiste à utiliser un champ de stockage privé pour définir et extraire la valeur de propriété. L'accesseur `get` retourne la valeur du champ privé et l'accesseur `set` peut effectuer une validation des données avant d'assigner une valeur au champ privé. Les deux accesseurs peuvent également effectuer des opérations de conversion ou de calcul sur les données avant de stocker ou retourner les données.

L'exemple suivant illustre ce modèle. Dans cet exemple, la classe `TimePeriod` représente un intervalle de temps. La classe stocke l'intervalle de temps en secondes, en interne, dans un champ privé nommé `_seconds`. L'utilisateur peut éventuellement spécifier l'intervalle de temps en heures à l'aide de la propriété en lecture-écriture `Hours`. Les deux accesseurs de propriété `get` et `set` effectuent ensuite la conversion nécessaire des heures en secondes. De plus, l'accesseur `set` valide les données et lève une exception ArgumentOutOfRangeException si le nombre d'heures n'est pas valide.

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Définitions de corps d'expression

Les accesseurs de propriété sont souvent des instructions sur une ligne qui ne font qu'assigner ou retourner le résultat d'une expression. Vous pouvez implémenter ces propriétés en tant que membres expression-bodied. Les définitions de corps d'expression se composent du symbole `=>` suivi de l'expression à assigner à la propriété ou à récupérer de la propriété.

À partir de C# 6, les propriétés en lecture seule peuvent implémenter l'accesseur `get` en tant que membre expression-bodied. Dans ce cas, le mot clé d'accesseur `get` et le mot clé `return` ne sont pas utilisés. L'exemple suivant implémente la propriété en lecture seule `Name` en tant que membre expression-bodied.

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Magnus", "Hedlund");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Magnus Hedlund
```

À compter de C# 7.0, les accesseurs `get` et `set` peuvent être implémentés comme membres expression-bodied. Dans ce cas, les mots clés `get` et `set` doivent être spécifiés. L'exemple suivant illustre l'utilisation de définitions de corps d'expression pour les deux accesseurs. Notez que le mot clé `return` n'est pas utilisé avec l'accesseur `get`.

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Propriétés implémentées automatiquement

Dans certains cas, les accesseurs de propriété `get` et `set` ne font qu'assigner une valeur à un champ de stockage, ou récupérer une valeur d'un champ de stockage, sans inclure de logique supplémentaire. En utilisant des propriétés implémentées automatiquement, vous simplifiez votre code, tout en laissant le compilateur C# fournir le champ de stockage de manière transparente.

Si une propriété a les accesseurs `get` et `set`, tous deux doivent être implémentés automatiquement. Vous définissez une propriété implémentée automatiquement à l'aide des mots clés `get` et `set` sans fournir d'implémentation. L'exemple suivant est identique à l'exemple précédent, sauf qu'il utilise les propriétés implémentées automatiquement `Name` et `Price`. Notez que l'exemple supprime également le constructeur paramétrable pour que les objets `SaleItem` soient initialisés avec un appel au constructeur sans paramètre et un initialiseur d'objet.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Sections connexes

- Utilisation de propriétés
- Propriétés de l'interface
- Comparaison entre propriétés et indexeurs
- Restriction d'accessibilité de l'accesseur
- Propriétés implémentées automatiquement

Spécification du langage C#

Pour plus d'informations, consultez Propriétés dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Utilisation de propriétés
- Indexeurs
- get, mot clé
- set, mot clé

Utilisation de propriétés (Guide de programmation C#)

02/11/2020 • 14 minutes to read • Edit Online

Les propriétés allient des caractéristiques des champs et des méthodes. Pour l'utilisateur d'un objet, une propriété s'apparente à un champ. Pour accéder à celle-ci, il doit utiliser la même syntaxe. Pour l'implémenteur d'une classe, une propriété est constituée d'un ou deux blocs de code, représentant un accesseur get et/ou un accesseur set. Le bloc de code correspondant à l'accesseur `get` est exécuté à la lecture de la propriété ; le bloc de code correspondant à l'accesseur `set` est exécuté au moment où une nouvelle valeur est assignée à la propriété. Une propriété sans accesseur `set` est considérée comme étant en lecture seule. Une propriété sans accesseur `get` est considérée comme étant en écriture seule. Une propriété qui possède les deux accesseurs est en lecture-écriture.

Contrairement aux champs, les propriétés ne sont pas classifiées en tant que variables. Par conséquent, vous ne pouvez pas passer une propriété en tant que paramètre ref ou out.

Les propriétés ont diverses utilisations : elles peuvent valider les données avant d'autoriser une modification ; elles peuvent exposer de manière transparente les données d'une classe quand celles-ci sont effectivement extraites d'une autre source, telle qu'une base de données ; elles peuvent effectuer une action quand des données sont modifiées, comme déclencher un événement ou modifier la valeur d'autres champs.

Les propriétés sont déclarées dans le bloc de classe en spécifiant le niveau d'accès du champ, suivi du type de la propriété, du nom de la propriété et d'un bloc de code qui déclare un accesseur `get` et/ou un accesseur `set`. Par exemple :

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

Dans cet exemple, `Month` est déclaré en tant que propriété pour permettre à l'accesseur `set` de vérifier que la valeur définie de `Month` se trouve bien entre 1 et 12. La propriété `Month` utilise un champ privé pour assurer le suivi de la valeur réelle. L'emplacement réel des données d'une propriété est souvent appelé « magasin de stockage » de la propriété. Il est courant que les propriétés utilisent des champs privés comme magasin de stockage. Le champ est marqué comme étant privé pour garantir qu'il ne peut être modifié qu'en appelant la propriété. Pour plus d'informations sur les restrictions d'accès public et privé, consultez Modificateurs d'accès.

Les propriétés implémentées automatiquement fournissent une syntaxe simplifiée pour des déclarations de propriété simples. Pour plus d'informations, consultez Propriétés implémentées automatiquement.

Accesseur get

Le corps de l'accesseur `get` ressemble à celui d'une méthode. Il doit retourner une valeur du type de la propriété. L'exécution de l'accesseur `get` revient à lire la valeur du champ. Par exemple, quand vous retournez la variable privée de l'accesseur `get` et que les optimisations sont activées, l'appel à la méthode de l'accesseur `get` est intégré (inline) par le compilateur, si bien il n'existe aucune surcharge d'appel de méthode. Cependant, une méthode d'accesseur `get` virtuelle ne peut pas être inline, car le compilateur ne sait pas au moment de la compilation quelle méthode peut être effectivement appelée au moment de l'exécution. Voici un exemple d'accesseur `get` qui retourne la valeur d'un champ privé `_name` :

```
class Person
{
    private string _name;  // the name field
    public string Name => _name;     // the Name property
}
```

Quand vous faites référence à la propriété (pas en tant que cible d'une assignation), l'accesseur `get` est appelé pour lire la valeur de la propriété. Par exemple :

```
Person person = new Person();
//...

System.Console.Write(person.Name);  // the get accessor is invoked here
```

L'accesseur `get` doit se terminer dans une instruction return ou throw et le contrôle ne peut pas déborder du corps de l'accesseur.

Modifier l'état de l'objet en utilisant l'accesseur `get` n'est pas un style de programmation approprié. Par exemple, l'accesseur suivant s'accompagne d'un effet secondaire, à savoir que l'état de l'objet est modifié chaque fois que le champ `_number` fait l'objet d'un accès.

```
private int _number;
public int Number => _number++; // Don't do this
```

L'accesseur `get` peut être utilisé pour retourner la valeur du champ ou pour la calculer et la retourner. Par exemple :

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

Dans le segment de code précédent, si vous n'assignez pas de valeur à la `Name` propriété, la valeur est retournée `NA` .

Accesseur Set

L'accesseur `set` ressemble à une méthode dont le type de retour est void. Il utilise un paramètre implicite nommé `value`, dont le type est celui de la propriété. Dans l'exemple ci-dessous, un accesseur `set` est ajouté à la propriété `Name` :

```
class Person
{
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

Quand vous assignez une valeur à la propriété, l'accesseur `set` est appelé en utilisant un argument qui fournit la nouvelle valeur. Par exemple :

```
Person person = new Person();
person.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(person.Name);  // the get accessor is invoked here
```

Il s'agit d'une erreur d'utiliser le nom de paramètre implicite, `value`, pour une déclaration de variable locale dans un accesseur `set`.

Notes

Les propriétés peuvent être marquées comme étant `public`, `private`, `protected`, `internal`, `protected internal` ou `private protected`. Ces modificateurs d'accès définissent comment les utilisateurs de la classe peuvent accéder à la propriété. Les accesseurs `get` et `set` d'une même propriété peuvent avoir des modificateurs d'accès différents. Par exemple, l'accesseur `get` peut être `public` pour autoriser l'accès en lecture seule en dehors du type, tandis que l'accesseur `set` peut être `private` ou `protected`. Pour plus d'informations, consultez Modificateurs d'accès.

Une propriété peut être déclarée en tant que propriété statique à l'aide du mot clé `static`. La propriété devient ainsi accessible à tout moment aux appelants, même s'il n'existe aucune instance de la classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Une propriété peut être marquée comme étant une propriété virtuelle à l'aide du mot clé virtual. Cela permet aux classes dérivées de substituer le comportement de la propriété à l'aide du mot clé override. Pour plus d'informations sur ces options, consultez Héritage.

Un propriété qui se substitue à une propriété virtuelle peut aussi être sealed, ce qui signifie que pour les classes dérivées, elle n'est plus virtuelle. Enfin, une propriété peut être déclarée comme étant abstract. Cela signifie qu'il n'existe aucune implémentation dans la classe et que les classes dérivées doivent écrire leur propre implémentation. Pour plus d'informations sur ces options, consultez Classes abstract et sealed et membres de classe.

> **NOTE**
> Il s'agit d'une erreur d'utiliser un modificateur virtual, abstract ou override sur un accesseur de propriété static.

Exemple

Cet exemple illustre les propriétés d'instance, statiques et en lecture seule. Il accepte le nom de l'employé à partir du clavier, incrémente `NumberOfEmployees` de 1 et affiche le nom et le numéro de l'employé.

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/
```

Exemple

Cet exemple montre comment accéder à une propriété d'une classe de base qui est masquée par une autre propriété qui porte le même nom dans une classe dérivée :

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

Voici les points importants de l'exemple précédent :

- La propriété `Name` de la classe dérivée masque la propriété `Name` de la classe de base. En pareil cas, le modificateur `new` est utilisé dans la déclaration de la propriété de la classe dérivée :

  ```
  public new string Name
  ```

- La conversion de type `(Employee)` est utilisée pour accéder à la propriété masquée de la classe de base :

  ```
  ((Employee)m1).Name = "Mary";
  ```

  Pour plus d'informations sur le masquage des membres, consultez new, modificateur.

Exemple

Dans cet exemple, deux classes, `Cube` et `Square`, implémentent une classe abstract, `Shape`, et remplacent sa propriété `Area` abstract. Notez l'utilisation du modificateur override sur les propriétés. Le programme accepte le côté (« side ») comme entrée et calcule les surfaces (« areas ») du carré (« square ») et du cube. De même, il

accepte la surface (« area ») comme entrée et calcule le côté (« side ») correspondant du carré (« square ») et du cube.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
```

```
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

Voir aussi

- Guide de programmation C#
- Propriétés
- Propriétés de l'interface
- Propriétés implémentées automatiquement

Propriétés d'interface (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Des propriétés peuvent être déclarées dans une interface. L'exemple suivant déclare un accesseur de propriété d'interface :

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Les propriétés d'interface n'ont généralement pas de corps. Les accesseurs indiquent si la propriété est en lecture-écriture, en lecture seule ou en écriture seule. Contrairement aux classes et aux structs, la déclaration des accesseurs sans corps ne déclare pas de propriété implémentée automatiquement. À compter de C# 8,0, une interface peut définir une implémentation par défaut pour les membres, y compris les propriétés. La définition d'une implémentation par défaut pour une propriété dans une interface est rare, car les interfaces ne peuvent pas définir des champs de données d'instance.

Exemple

Dans cet exemple, l'interface `IEmployee` a une propriété en lecture-écriture, `Name`, et une propriété en lecture seule, `Counter`. La classe `Employee` implémente l'interface `IEmployee` et utilise ces deux propriétés. Le programme lit le nom d'un nouvel employé et le nombre actuel d'employés et affiche le nom de l'employé et le nombre d'employés calculé.

Vous pouvez utiliser le nom qualifié complet de la propriété, qui fait référence à l'interface dans laquelle le membre est déclaré. Par exemple :

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

L'exemple précédent illustre l' implémentation d'interface explicite. Par exemple, si la classe `Employee` implémente deux interfaces, `ICitizen` et `IEmployee`, et que les deux interfaces ont la même propriété `Name`, l'implémentation de membre d'interface explicite est nécessaire. Autrement dit, la déclaration de propriété suivante :

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implémente la propriété `Name` dans l'interface `IEmployee`, alors que la déclaration suivante :

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implémente la propriété `Name` dans l'interface `ICitizen`.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

`210 Hazem Abolrous`

Exemple de sortie

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

Voir aussi

- Guide de programmation C#
- Propriétés
- Utilisation de propriétés
- Comparaison entre propriétés et indexeurs
- Indexeurs
- Interfaces

Restriction d'accessibilité de l'accesseur (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Les parties get et set d'une propriété ou d'un indexeur sont appelées *accesseurs*. Par défaut, ces accesseurs ont la visibilité ou le niveau d'accès de la propriété ou de l'indexeur auquel ils appartiennent. Pour plus d'informations, consultez Niveaux d'accessibilité. Toutefois, il peut parfois s'avérer utile de restreindre l'accès à l'un de ces accesseurs. En général, cela implique de restreindre l'accessibilité de l'accesseur `set`, tout en gardant l'accesseur `get` publiquement accessible. Par exemple :

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

Dans cet exemple, une propriété appelée `Name` définit un accesseur `get` et `set`. L'accesseur `get` reçoit le niveau d'accessibilité de la propriété elle-même, `public` dans le cas présent, alors que l'accesseur `set` est restreint explicitement par l'application du modificateur d'accès protected à l'accesseur lui-même.

Restrictions sur les modificateurs d'accès sur les accesseurs

L'utilisation de modificateurs d'accesseurs sur les propriétés ou les indexeurs est soumise aux conditions suivantes :

- Vous ne pouvez pas utiliser de modificateurs d'accesseur sur une interface ou sur une implémentation de membre d'interface explicite.
- Vous pouvez utiliser les modificateurs d'accesseur uniquement si la propriété ou l'indexeur dispose à la fois d'accesseurs `set` et `get`. Dans ce cas, le modificateur est autorisé uniquement sur l'un des deux accesseurs.
- Si la propriété ou l'indexeur possède un modificateur override, le modificateur d'accesseur doit correspondre à l'accesseur de l'accesseur substitué, le cas échant.
- Le niveau d'accessibilité sur l'accesseur doit être plus restrictif que le niveau d'accessibilité sur la propriété ou l'indexeur eux-mêmes.

Modificateurs d'accès sur les accesseurs de substitution

Quand vous substituez une propriété ou un indexeur, les accesseurs remplacés doivent être accessibles au code de substitution. Par ailleurs, l'accessibilité de la propriété/l'indexeur et de ses accesseurs doivent correspondre à la propriété/l'indexeur et à ses accesseurs substitués. Par exemple :

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Implémentation des interfaces

Quand vous utilisez un accesseur pour implémenter une interface, l'accesseur peut ne pas avoir de modificateur d'accès. Toutefois, si vous implémentez l'interface à l'aide d'un accesseur, tel que `get`, l'autre accesseur peut avoir un modificateur d'accès, comme dans l'exemple suivant :

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Domaine d'accessibilité de l'accesseur

Si vous utilisez un modificateur d'accès sur l'accesseur, le domaine d'accessibilité de l'accesseur est déterminé par ce modificateur.

Si vous n'avez pas utilisé un modificateur d'accès sur l'accesseur, le domaine d'accessibilité de l'accesseur est déterminé par le niveau d'accessibilité de la propriété ou de l'indexeur.

Exemple

L'exemple suivant contient trois classes, `BaseClass`, `DerivedClass` et `MainClass`. Il y a deux propriétés sur `BaseClass`, `Name` et `Id` sur les deux classes. L'exemple montre comment la propriété `Id` sur `DerivedClass` peut être masquée par la propriété `Id` sur `BaseClass` quand vous utilisez un modificateur d'accès restrictif tel que protected ou private. Par conséquent, quand vous affectez des valeurs à cette propriété, la propriété sur la classe `BaseClass` est appelée à la place. Le remplacement du modificateur d'accès par public rend la propriété accessible.

L'exemple montre également qu'un modificateur d'accès restrictif, tel que `private` ou `protected`, sur l'accesseur `set` de la propriété `Name` dans `DerivedClass` empêche l'accès à l'accesseur et génère une erreur quand vous lui affectez une valeur.

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}
```

```
class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Commentaires

Notez que si vous remplacez la déclaration `new private string Id` par `new public string Id`, vous obtenez la sortie suivante :

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

Voir aussi

- Guide de programmation C#
- Propriétés
- Indexeurs
- Modificateurs d'accès

Comment déclarer et utiliser les propriétés en lecture/écriture (Guide de programmation C#)

16/12/2020 • 4 minutes to read • Edit Online

Les propriétés offrent la commodité des membres de données publics sans les risques liés à un accès non protégé, non contrôlé et non vérifié aux données d'un objet. Cela se fait au moyen d'*accesseurs*, lesquels sont des méthodes spéciales qui affectent et récupèrent des valeurs du membre de données sous-jacent. L'accesseur set permet aux membres de données d'être affectés, et l'accesseur get récupère des valeurs de membres de données.

L'exemple suivant montre une classe `Person` qui possède deux propriétés : `Name` (string) et `Age` (int). Étant donné que les deux propriétés fournissent des accesseurs `get` et `set`, elles sont considérées comme des propriétés en lecture/écriture.

Exemple

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
```

```
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Programmation fiable

Dans l'exemple précédent, les propriétés `Name` et `Age` sont publiques, et incluent un accesseur `get` et un accesseur `set`. Cela permet à n'importe quel objet de lire et d'écrire ces propriétés. Toutefois, il est parfois souhaitable d'exclure l'un des accesseurs. L'omission de l'accesseur `set`, par exemple, met la propriété en lecture seule :

```
public string Name
{
    get
    {
        return _name;
    }
}
```

Vous pouvez également exposer publiquement un accesseur mais rendre l'autre private ou protected. Pour plus d'informations, consultez Accessibilité de l'accesseur asymétrique.

Une fois les propriétés déclarées, vous pouvez les utiliser comme s'il s'agissait de champs de la classe. Cela permet une syntaxe très naturelle pour obtenir ou définir la valeur d'une propriété, comme dans les instructions suivantes :

```
person.Name = "Joe";
person.Age = 99;
```

Notez qu'une variable `value` spéciale est disponible dans la méthode `set` d'une propriété. Cette variable contient la valeur que l'utilisateur a spécifiée, par exemple :

```
_name = value;
```

Notez la syntaxe correcte pour incrémenter la propriété `Age` sur un objet `Person` :

```
person.Age += 1;
```

Si des méthodes `set` et `get` distinctes ont été utilisées pour modeler des propriétés, le code équivalent peut avoir la forme suivante :

```
person.SetAge(person.GetAge() + 1);
```

La méthode `ToString` est substituée dans cet exemple :

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Vous pouvez remarquer que `ToString` n'est pas utilisée de façon explicite dans le programme. Elle est appelée par défaut par les appels `WriteLine`.

Voir aussi

- Guide de programmation C#
- Propriétés
- Classes et structs

Propriétés implémentées automatiquement (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

En C# 3.0 et versions ultérieures, les propriétés implémentées automatiquement rendent la déclaration de propriété plus concise quand aucune logique supplémentaire n'est requise dans les accesseurs de propriété. Elles permettent également au code client de créer des objets. Quand vous déclarez une propriété comme indiqué dans l'exemple suivant, le compilateur crée un champ de stockage privé et anonyme uniquement accessible via les accesseurs `get` et `set` de la propriété.

Exemple

L'exemple suivant montre une classe simple qui a des propriétés implémentées automatiquement :

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

Vous ne pouvez pas déclarer les propriétés implémentées automatiquement dans les interfaces. Les propriétés implémentées automatiquement déclarent un champ de stockage d'instance privée, et les interfaces peuvent ne pas déclarer de champs d'instance. La déclaration d'une propriété dans une interface sans définition de corps déclare une propriété avec des accesseurs qui doivent être implémentés par chaque type qui implémente cette interface.

En C# 6 et versions ultérieures, vous pouvez initialiser des propriétés implémentées automatiquement de la même façon que des champs :

```
public string FirstName { get; set; } = "Jane";
```

La classe qui est illustrée dans l'exemple précédent est mutable. Le code client peut modifier les valeurs dans les objets après la création. Dans les classes complexes qui contiennent un comportement significatif (méthodes) ainsi que des données, il est souvent nécessaire d'avoir des propriétés publiques. Toutefois, pour les petites classes ou les petits structs qui encapsulent simplement un ensemble de valeurs (données) et qui ont peu de comportements ou aucun, vous devez rendre les objets immuables en déclarant l'accesseur set comme étant privé (immuables pour les consommateurs) ou en déclarant uniquement un accesseur get (immuables partout sauf dans le constructeur). Pour plus d'informations, consultez comment implémenter une classe Lightweight avec des propriétés implémentées automatiquement.

Voir aussi

- Propriétés
- Modificateurs

Comment implémenter une classe Lightweight avec des propriétés implémentées automatiquement (Guide de programmation C#)

16/12/2020 • 4 minutes to read • Edit Online

Cet exemple montre comment créer une classe légère immuable qui sert uniquement à encapsuler un jeu de propriétés implémentées automatiquement. Utilisez ce type de construction à la place d'un struct quand vous devez utiliser une sémantique de type de référence.

Vous pouvez rendre une propriété immuable de deux manières :

- Vous pouvez déclarer l'accesseur set comme étant private. La propriété peut uniquement être définie dans le type, mais elle est immuable pour les consommateurs.

  Quand vous déclarez un accesseur `set` privé, vous ne pouvez pas utiliser un initialiseur d'objet pour initialiser la propriété. Vous devez utiliser un constructeur ou une méthode de fabrique.

- Vous pouvez déclarer uniquement l'accesseur Get , ce qui rend la propriété immuable partout sauf dans le constructeur du type.

L'exemple suivant montre comment une propriété avec uniquement l'accesseur Get diffère de celle avec Get et Private Set.

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress
}
```

Exemple

L'exemple suivant montre deux façons d'implémenter une classe immuable qui possède des propriétés implémentées automatiquement. Chaque façon déclare l'une des propriétés avec un `set` privé et l'autre avec un `get` uniquement. La première classe utilise un constructeur uniquement pour initialiser les propriétés et la deuxième classe utilise une méthode de fabrique statique qui appelle un constructeur.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
```

```
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                              "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                         select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();
```

```
        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

Le compilateur crée des champs de stockage pour chaque propriété implémentée automatiquement. Les champs ne sont pas accessibles directement à partir du code source.

Voir aussi

- Propriétés
- modélis
- Initialiseurs d'objets et de collections

Méthodes (Guide de programmation C#)

02/11/2020 • 18 minutes to read • Edit Online

Une méthode est un bloc de code qui contient une série d'instructions. Un programme provoque l'exécution des instructions en appelant la méthode et en spécifiant les éventuels arguments de méthode requis. En C#, chaque instruction exécutée est effectuée dans le contexte d'une méthode. La `Main` méthode est le point d'entrée de chaque application C# et elle est appelée par le Common Language Runtime (CLR) au démarrage du programme.

> **NOTE**
> Cet article décrit les méthodes nommées. Pour plus d'informations sur les fonctions anonymes, consultez Fonctions anonymes.

Signatures de méthode

Les méthodes sont déclarées dans une classe, un structou une interface en spécifiant le niveau d'accès comme `public` ou `private` , les modificateurs facultatifs tels que `abstract` ou `sealed` , la valeur de retour, le nom de la méthode et tous les paramètres de méthode. Ces parties forment ensemble la signature de la méthode.

> **IMPORTANT**
> Un type de retour d'une méthode ne fait pas partie de la signature de la méthode à des fins de surcharge de méthode. Toutefois, il fait partie de la signature de la méthode lors de la détermination de la compatibilité entre un délégué et la méthode vers laquelle il pointe.

Les paramètres de méthode sont placés entre parenthèses et séparés par des virgules. Des parenthèses vides indiquent que la méthode ne requiert aucun paramètre. Cette classe contient quatre méthodes :

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Accès aux méthodes

L'appel d'une méthode sur un objet revient à accéder à un champ. Après le nom de l'objet, ajoutez un point, le nom de la méthode et les parenthèses. Les arguments sont répertoriés entre les parenthèses et séparés par des virgules. Les méthodes de la classe `Motorcycle` peuvent donc être appelées comme dans l'exemple

suivant :

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Paramètres de méthode et arguments

La définition de la méthode spécifie les noms et types des paramètres requis. Quand le code appelant appelle la méthode, il fournit des valeurs concrètes appelées arguments pour chaque paramètre. Les arguments doivent être compatibles avec le type de paramètre, mais le nom d'argument (le cas échéant) utilisé dans le code appelant ne doit pas nécessairement être le même que le paramètre nommé défini dans la méthode. Par exemple :

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passage par référence et passage par valeur

Par défaut, lorsqu'une instance d'un type valeur est passée à une méthode, sa copie est passée au lieu de l'instance elle-même. Par conséquent, les modifications apportées à l'argument n'ont aucun effet sur l'instance d'origine dans la méthode d'appel. Pour passer une instance de type valeur par référence, utilisez le

`ref` mot clé. Pour plus d'informations, consultez Passage de paramètres de type valeur .

Quand un objet d'un type référence est passé à une méthode, une référence à l'objet est passée. Autrement dit, la méthode ne reçoit pas l'objet lui-même mais un argument qui indique l'emplacement de l'objet. Si vous modifiez un membre de l'objet à l'aide de cette référence, la modification est répercutée dans l'argument de la méthode d'appel, même si vous passez l'objet par valeur.

Vous créez un type référence à l'aide du `class` mot clé, comme le montre l'exemple suivant :

```
public class SampleRefType
{
    public int value;
}
```

Maintenant, si vous passez un objet basé sur ce type à une méthode, une référence à l'objet est passée. L'exemple suivant passe un objet de type `SampleRefType` à la méthode `ModifyObject` :

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

L'exemple produit essentiellement la même chose que l'exemple précédent, dans la mesure où il passe un argument par valeur à une méthode. Mais, puisqu'un type référence est utilisé, le résultat est différent. La modification apportée dans `ModifyObject` au champ `value` du paramètre, `obj`, modifie également le champ `value` de l'argument, `rt`, dans la méthode `TestRefType` . La méthode `TestRefType` affiche 33 en tant que sortie.

Pour plus d'informations sur le passage de types référence par référence et par valeur, consultez Passage de paramètres de type référence et Types référence.

Valeurs retournées

Les méthodes peuvent retourner une valeur à l'appelant. Si le type de retour, le type répertorié avant le nom de la méthode, n'est pas `void`, la méthode peut retourner la valeur à l'aide du mot clé `return` . Une instruction avec le mot clé `return` suivi d'une valeur qui correspond au type de retour retourne cette valeur à l'appelant de la méthode.

La valeur peut être retournée à l'appelant par valeur ou, à compter de C# 7.0, par référence. Les valeurs sont retournées à l'appelant par référence si le mot clé `ref` est utilisé dans la signature de méthode et s'il suit chaque mot clé `return`. Par exemple, la signature de méthode suivante et l'instruction de retour indiquent que la méthode retourne un nom de variable `estDistance` par référence à l'appelant.

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

Le mot clé `return` arrête également l'exécution de la méthode. Si le type de retour est `void`, une instruction

`return` sans valeur est quand même utile pour arrêter l'exécution de la méthode. Sans le mot clé `return` , la méthode arrête de s'exécuter quand elle atteint la fin du bloc de code. Les méthodes dotées d'un type de retour non void doivent utiliser le mot clé `return` pour retourner une valeur. Par exemple, ces deux méthodes utilisent le mot clé `return` pour retourner des entiers :

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Pour utiliser une valeur retournée à partir d'une méthode, la méthode d'appel peut utiliser l'appel de méthode proprement dit partout où une valeur du même type peut suffire. Vous pouvez également affecter la valeur de retour à une variable. Par exemple, les deux exemples de code suivants remplissent le même objectif :

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

L'utilisation d'une variable locale, dans cet exemple, `result`, pour stocker une valeur est facultative. Elle peut favoriser la lisibilité du code ou s'avérer nécessaire si vous avez besoin de stocker la valeur d'origine de l'argument pour la portée entière de la méthode.

Pour utiliser une valeur retournée par référence à partir d'une méthode, vous devez déclarer une variable ref local si vous avez l'intention de modifier sa valeur. Par exemple, si la méthode `Planet.GetEstimatedDistance` retourne une valeur Double par référence, vous pouvez la définir en tant que variable ref local avec du code semblable à celui-ci :

```
ref int distance = plant
```

Le retour d'un tableau multidimensionnel à partir d'une méthode, `M`, qui modifie le contenu du tableau n'est pas nécessaire si la fonction appelante a passé le tableau à `M`. Vous pouvez retourner le tableau obtenu à partir de `M` pour bénéficier d'un style approprié ou d'un flux fonctionnel de valeurs, mais cela n'est pas nécessaire, car C# passe tous les types de référence par valeur, et la valeur d'une référence de tableau est le pointeur qui désigne le tableau. Dans la méthode `M` , toute modification apportée au contenu du tableau est observable par tout code qui possède une référence au tableau, comme indiqué dans l'exemple suivant :

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Pour plus d'informations, consultez return.

Méthodes Async

La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels explicites ni fractionner manuellement votre code entre plusieurs méthodes ou expressions lambda.

Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression await dans la méthode async, il retourne à l'appelant, et la progression dans la méthode est interrompue jusqu'à ce que la tâche attendue se termine. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode.

> **NOTE**
> Une méthode Async retourne à l'appelant lorsqu'elle rencontre le premier objet attendu qui n'est pas encore terminé ou qu'elle atteint la fin de la méthode Async, selon ce qui se produit en premier.

Une méthode async peut avoir un type de retour Task<TResult>, Taskou void. Le type de retour void est principalement utilisé pour définir les gestionnaires d'événements, où un type de retour void est requis. Une méthode async qui retourne void ne peut pas être attendue, et l'appelant d'une méthode retournant void ne peut intercepter aucune exception levée par la méthode.

Dans l'exemple suivant, `DelayAsync` est une méthode async dont le type de retour est Task<TResult>. `DelayAsync` a une instruction `return` qui retourne un entier. Ainsi, la déclaration de méthode de `DelayAsync` doit avoir un type de retour de `Task<int>`. Étant donné que le type de retour est `Task<int>`, l'évaluation de l'expression `await` dans `DoSomethingAsync` produit un entier, comme le montre l'instruction suivante : `int result = await delayTask`.

La `Main` méthode est un exemple de méthode Async dont le type de retour est Task . Elle accède à la `DoSomethingAsync` méthode et, comme elle est exprimée par une seule ligne, elle peut omettre les `async` `await` Mots clés et. Étant donné que `DoSomethingAsync` est une méthode async, la tâche pour l'appel à `DoSomethingAsync` doit être attendue, comme le montre l'instruction suivante : `await DoSomethingAsync();`.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Une méthode async ne peut pas déclarer de paramètres ref ou out , mais elle peut appeler des méthodes qui comportent ces paramètres.

Pour plus d'informations sur les méthodes Async, consultez programmation asynchrone avec Async et await et Async (types de retour).

Définitions de corps d'expression

Il est courant d'avoir des définitions de méthode qui retournent tout simplement le résultat d'une expression immédiatement, ou qui ont une seule instruction en tant que corps de la méthode. Il existe un raccourci de syntaxe pour définir de telles méthodes en utilisant `=>` :

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si la méthode retourne `void` ou est une méthode async, alors le corps de la méthode doit être une expression d'instruction (même chose avec les expressions lambda). En ce qui concerne les propriétés et indexeurs, ils doivent être en lecture seule et vous n'utilisez pas le mot clé d'accesseur `get` .

Iterators

Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction yield return est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution redémarre à partir de cet emplacement au prochain appel de l'itérateur.

Vous appelez un itérateur depuis le code client en utilisant une instruction foreach.

Le type de retour d'un itérateur peut être IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Pour plus d'informations, consultez itérateurs.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Modificateurs d'accès
- Classes statiques et membres de classe statique
- Héritage
- Classes abstract et sealed et membres de classe
- params
- renvoi
- à
- ref
- Passer des paramètres

Fonctions locales (Guide de programmation C#)

02/11/2020 • 20 minutes to read • Edit Online

À compter de C# 7.0, C# prend en charge les *fonctions locales*. Les fonctions locales sont des méthodes privées d'un type qui sont imbriqués dans un autre membre. Elles ne peuvent être appelées qu'à partir de leur membre conteneur. Les fonctions locales peuvent être déclarées et appelées dans et à partir des éléments suivants :

- Méthodes, tout particulièrement les méthodes iterator et async
- Constructeurs
- Accesseurs de propriété
- Accesseurs d'événement
- Méthodes anonymes
- Expressions lambda
- Finaliseurs
- Autres fonctions locales

En revanche, les fonctions locales ne peuvent pas être déclarées à l'intérieur d'un membre expression-bodied.

> **NOTE**
> Dans certains cas, vous pouvez utiliser une expression lambda pour implémenter la fonctionnalité également prise en charge par une fonction locale. Pour une comparaison, consultez fonctions locales et expressions lambda.

Les fonctions locales permettent de clarifier l'objectif de votre code. Toute personne lisant votre code peut voir que la méthode ne peut pas être appelée, sauf par la méthode conteneur. Pour les projets d'équipe, elles empêchent aussi à un autre développeur d'appeler par inadvertance la méthode directement à partir d'un autre emplacement dans la classe ou le struct.

Syntaxe des fonctions locales

Une fonction locale est définie en tant que méthode imbriquée à l'intérieur d'un membre conteneur. Sa définition présente la syntaxe suivante :

```
<modifiers> <return-type> <method-name> <parameter-list>
```

Vous pouvez utiliser les modificateurs suivants avec une fonction locale :

- `async`
- `unsafe`
- `static` (en C# 8,0 et versions ultérieures). Une fonction locale statique ne peut pas capturer les variables locales ou l'état de l'instance.
- `extern` (en C# 9,0 et versions ultérieures). Une fonction locale externe doit être `static` .

Toutes les variables locales définies dans le membre conteneur, y compris ses paramètres de méthode, sont accessibles dans une fonction locale non statique.

Contrairement à une définition de méthode, une définition de fonction locale ne peut pas inclure le modificateur d'accès au membre. Comme toutes les fonctions locales sont privées, l'inclusion d'un modificateur d'accès, tel que le mot clé `private`, génère l'erreur de compilateur CS0106 : « Le modificateur « private » n'est pas valide pour cet

élément ».

L'exemple suivant définit une fonction locale nommée `AppendPathSeparator` qui est privée pour une méthode nommée `GetText` :

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

À compter de C# 9,0, vous pouvez appliquer des attributs à une fonction locale, à ses paramètres et à ses paramètres de type, comme le montre l'exemple suivant :

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

L'exemple précédent utilise un attribut spécial pour aider le compilateur dans une analyse statique dans un contexte Nullable.

Fonctions locales et exceptions

L'une des caractéristiques intéressantes des fonctions locales est qu'elles permettent l'affichage immédiat des exceptions. Pour les itérateurs de méthode, les exceptions apparaissent uniquement au moment où la séquence retournée est énumérée, et non à la récupération de l'itérateur. Pour les méthodes async, les exceptions levées dans une méthode async sont observées quand la tâche retournée est attendue.

L'exemple suivant définit une `OddSequence` méthode qui énumère les nombres impairs dans une plage spécifiée. Sachant qu'elle passe à la méthode d'énumérateur `OddSequence` un nombre supérieur à 100, la méthode lève une exception ArgumentOutOfRangeException. Comme le montre la sortie de l'exemple, l'exception apparaît uniquement au moment d'itérer les nombres, et non à la récupération de l'énumérateur.

```
using System;
using System.Collections.Generic;

public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
(Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in
IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11
```

Si vous placez la logique d'itérateur dans une fonction locale, les exceptions de validation d'argument sont levées lorsque vous récupérez l'énumérateur, comme le montre l'exemple suivant :

```
using System;
using System.Collections.Generic;

public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
(Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Vous pouvez utiliser des fonctions locales de la même façon avec les opérations asynchrones. Exceptions levées dans une surface de méthode Async lorsque la tâche correspondante est attendue. Les fonctions locales permettent à votre code d'échouer rapidement et à votre exception d'être à la fois levée et observée de manière synchrone.

L'exemple suivant utilise une méthode asynchrone nommée `GetMultipleAsync` visant à marquer une pause pendant un nombre défini de secondes et retourner une valeur qui est un multiple aléatoire de ce nombre de secondes. Le délai maximal est de 5 secondes ; une exception ArgumentOutOfRangeException est obtenue si la valeur est supérieure à 5. Comme le montre l'exemple suivant, l'exception levée lorsqu'une valeur de 6 est passée à la `GetMultipleAsync` méthode est observée uniquement lorsque la tâche est attendue.

```
using System;
using System.Threading.Tasks;

public class AsyncWithoutLocalExample
{
   public static async Task Main()
   {
      var t = GetMultipleAsync(6);
      Console.WriteLine("Got the task");

      var result = await t;  // line 11
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static async Task<int> GetMultipleAsync(int delayInSeconds)
   {
      if (delayInSeconds < 0 || delayInSeconds > 5)
         throw new ArgumentOutOfRangeException(nameof(delayInSeconds), "Delay cannot exceed 5 seconds.");

      await Task.Delay(delayInSeconds * 1000);
      return delayInSeconds * new Random().Next(2,10);
   }
}
// The example displays the output like this:
//
// Got the task
// Unhandled exception. System.ArgumentOutOfRangeException: Delay cannot exceed 5 seconds. (Parameter
'delayInSeconds')
//   at AsyncWithoutLocalExample.GetMultipleAsync(Int32 delayInSeconds) in AsyncWithoutLocal.cs:line 18
//   at AsyncWithoutLocalExample.Main() in AsyncWithoutLocal.cs:line 11
```

Comme avec l'itérateur de méthode, vous pouvez refactoriser l'exemple précédent et placer le code de l'opération asynchrone dans une fonction locale. Comme le montre la sortie de l'exemple suivant, le ArgumentOutOfRangeException est levé dès que la `GetMultiple` méthode est appelée.

```
using System;
using System.Threading.Tasks;

public class AsyncWithLocalExample
{
   public static async Task Main()
   {
      var t = GetMultiple(6);  // line 8
      Console.WriteLine("Got the task");

      var result = await t;
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static Task<int> GetMultiple(int delayInSeconds)
   {
      if (delayInSeconds < 0 || delayInSeconds > 5)
         throw new ArgumentOutOfRangeException(nameof(delayInSeconds), "Delay cannot exceed 5 seconds.");

      return GetValueAsync();

      async Task<int> GetValueAsync()
      {
         await Task.Delay(delayInSeconds * 1000);
         return delayInSeconds * new Random().Next(2,10);
      }
   }
}
// The example displays the output like this:
//
// Unhandled exception. System.ArgumentOutOfRangeException: Delay cannot exceed 5 seconds. (Parameter
'delayInSeconds')
//   at AsyncWithLocalExample.GetMultiple(Int32 delayInSeconds) in AsyncWithLocal.cs:line 18
//   at AsyncWithLocalExample.Main() in AsyncWithLocal.cs:line 8
```

Fonctions locales et expressions lambda

À première vue, les fonctions locales et les expressions lambda sont très similaires. Souvent, le choix d'utiliser des expressions lambda ou des fonctions locales est une question de style et de préférences personnelles. Toutefois, il existe de réelles différences qui vous feront utiliser les unes ou les autres et que vous devez connaître.

Examinons les différences entre l'implémentation de l'algorithme factoriel avec une fonction locale et une expression lambda. Voici la version à l'aide d'une fonction locale :

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

Cette version utilise des expressions lambda :

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Dénomination

Les fonctions locales sont nommées explicitement comme les méthodes. Les expressions lambda sont des méthodes anonymes qui doivent être assignées à des variables d'un `delegate` type, généralement des `Action` `Func` types ou. Lorsque vous déclarez une fonction locale, le processus est semblable à l'écriture d'une méthode normale ; vous déclarez un type de retour et une signature de fonction.

Signatures de fonctions et types d'expression lambda

Les expressions lambda s'appuient sur le type de la `Action` / `Func` variable qu'elles sont assignées pour déterminer les types d'arguments et de retour. Dans les fonctions locales, étant donné que la syntaxe est très similaire à l'écriture d'une méthode normale, les types d'argument et le type de retour font déjà partie de la déclaration de fonction.

Assignation définie

Les expressions lambda sont des objets qui sont déclarés et assignés au moment de l'exécution. Pour qu'une expression lambda soit utilisée, elle doit être assignée de manière définitive : la `Action` / `Func` variable à laquelle elle sera assignée doit être déclarée et l'expression lambda assignée à celle-ci. Notez que `LambdaFactorial` doit déclarer et initialiser l'expression lambda `nthFactorial` avant de la définir. Si ce n'est pas le cas, cela entraîne une erreur de compilation due au fait que vous référencez `nthFactorial` avant de lui affecter une valeur.

Les fonctions locales sont définies au moment de la compilation. Comme elles ne sont pas affectées à des variables, elles peuvent être référencées à partir de n'importe quel emplacement de code **où elles se trouvent dans la portée**; dans notre premier exemple `LocalFunctionFactorial` , nous pourrions déclarer notre fonction locale soit au-dessus soit au-dessous de l' `return` instruction et ne déclenchent aucune erreur du compilateur.

Ces différences font que les algorithmes récursifs sont plus faciles à créer en utilisant des fonctions locales. Vous pouvez déclarer et définir une fonction locale qui s'appelle elle-même. Les expressions lambda doivent être déclarées et une valeur par défaut doit leur être affectée avant qu'elles puissent être réaffectées à un corps référençant la même expression lambda.

Implémentation en tant que délégué

Les expressions lambda sont converties en délégués lorsqu'elles sont déclarées. Les fonctions locales sont plus flexibles dans la mesure où elles peuvent être écrites comme une méthode traditionnelle *ou* en tant que délégué. Les fonctions locales sont converties en délégués uniquement lorsqu'elles sont ***utilisées*** en tant que délégué.

Si vous déclarez une fonction locale et la référencez uniquement en l'appelant comme une méthode, elle ne sera pas convertie en délégué.

Capture de variables

Les règles d' assignation définie affectent également toutes les variables capturées par la fonction locale ou l'expression lambda. Le compilateur peut effectuer une analyse statique qui permet aux fonctions locales d'affecter de manière définitive des variables capturées dans la portée englobante. Prenons l'exemple suivant :

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

Le compilateur peut déterminer que `LocalFunction` affecte `y` de manière définitive lorsqu'elle est appelée. Dans la mesure où `LocalFunction` est appelée avant l'instruction `return`, `y` est affecté de manière définitive à l'instruction `return`.

Notez que lorsqu'une fonction locale capture des variables dans la portée englobante, la fonction locale est implémentée en tant que type délégué.

Allocations de tas

En fonction de leur utilisation, les fonctions locales peuvent éviter les allocations de tas qui sont toujours nécessaires pour les expressions lambda. Si une fonction locale n'est jamais convertie en délégué, et si aucune des variables capturées par la fonction locale n'est capturée par d'autres expressions lambda ou fonctions locales converties en délégués, le compilateur peut éviter les allocations de tas.

Penchons-nous sur cet exemple asynchrone :

```
public Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    };

    return longRunningWorkImplementation();
}
```

La fermeture de cette expression lambda contient les variables `address`, `index` et `name`. Dans le cas des fonctions locales, l'objet qui implémente la fermeture peut être un type `struct`. Ce type de struct serait transmis par référence à la fonction locale. Cette différence d'implémentation évite une allocation.

L'instanciation nécessaire pour les expressions lambda signifie des allocations de mémoire supplémentaires, qui peuvent être un facteur influençant les performances dans les chemins de code critiques au niveau du temps. Les fonctions locales n'entraînent pas cette charge supplémentaire. Dans l'exemple ci-dessus, la version des fonctions locales a deux allocations moins nombreuses que la version de l'expression lambda.

Si vous savez que votre fonction locale ne sera pas convertie en un délégué et qu'aucune des variables capturées par celle-ci n'est capturée par d'autres expressions lambda ou fonctions locales converties en délégués, vous pouvez vous assurer que votre fonction locale évite d'être allouée sur le tas en la déclarant comme une `static` fonction locale. Notez que cette fonctionnalité est disponible dans C# 8,0 et versions ultérieures.

> **NOTE**
> L'équivalent de cette méthode avec une fonction locale fait aussi appel à une classe pour la fermeture. Le fait que la fermeture d'une fonction locale soit implémentée en tant que `class` ou `struct` est un détail d'implémentation. Une fonction locale peut utiliser un type `struct` contrairement à une expression lambda qui utilise toujours un type `class`.

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Utilisation du `yield` mot clé

Ultime avantage non décrit dans cet exemple : les fonctions locales peuvent être implémentées en tant qu'itérateurs, en utilisant la syntaxe `yield return` pour produire une séquence de valeurs.

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

L' `yield return` instruction n'est pas autorisée dans les expressions lambda, consultez Erreur du compilateur CS1621.

Alors que les fonctions locales peuvent sembler redondantes par rapport aux expressions lambda, elles ont en réalité des objectifs différents et des utilisations différentes. Les fonctions locales sont plus efficaces dans le cas où vous voulez écrire une fonction qui est appelée seulement dans le contexte d'une autre méthode.

Voir aussi

- Méthodes

Retours ref et variables locales ref

18/03/2020 • 13 minutes to read • Edit Online

À compter de C# 7.0, C# prend en charge les valeurs de retour de référence (retours ref). Une valeur de retour de référence permet à une méthode de retourner à un appelant une référence à une variable, plutôt qu'à une valeur. L'appelant peut alors choisir de traiter la variable retournée comme si elle était retournée par valeur ou par référence. L'appelant peut créer une nouvelle variable qui est elle-même une référence à la valeur retournée, appelée variable locale ref.

Qu'est-ce qu'une valeur de retour de référence ?

La plupart des développeurs sont familiarisés avec le passage d'un argument à une méthode appelée *par référence*. La liste d'arguments d'une méthode appelée inclut une variable passée par référence. Chaque modification de sa valeur par la méthode appelée est respectée par l'appelant. Une *valeur de retour de référence* signifie qu'une méthode retourne une *référence* (ou un alias) à une variable. L'étendue de cette variable doit inclure la méthode. La durée de vie de cette variable doit s'étendre au-delà du retour de la méthode. Des modifications apportées par l'appelant à la valeur de retour de la méthode portent sur la variable qui est retournée par la méthode.

La déclaration qu'une méthode retourne une *valeur de retour de référence* indique que la méthode retourne un alias vers une variable. L'intention de conception est souvent que le code appelant ait accès à cette variable à travers l'alias, et qu'il puisse également la modifier. C'est pourquoi les méthodes de retour par référence ne peuvent pas avoir le type de retour `void`.

Certaines restrictions s'appliquent à l'expression qu'une méthode peut retourner comme valeur de retour de référence. Les restrictions sont les suivantes :

- La valeur de retour doit avoir une durée de vie qui s'étend au-delà de l'exécution de la méthode. En d'autres termes, il ne peut pas s'agir d'une variable locale dans la méthode qui la retourne. Il peut s'agir d'un champ d'instance ou statique d'une classe, ou bien un argument passé à la méthode. Une tentative de retour d'une variable locale génère l'erreur du compilateur CS8168, « Impossible de retourner la variable locale 'obj' par référence, car il ne s'agit pas d'une variable locale de référence ».
- La valeur de retour ne peut pas être le littéral `null`. Le retour de `null` génère l'erreur de compilateur CS8156, « Impossible d'utiliser une expression dans ce contexte, car elle ne peut pas être retournée par référence ».

  Une méthode avec un retour d'arbitre peut retourner un alias à une variable dont la valeur est actuellement la valeur nulle (non corroborée) ou un type de valeur nul pour un type de valeur.
- La valeur de retour ne peut pas être une constante, un membre d'énumération, la valeur de retour par valeur d'une propriété, ou une méthode d'une `class` ou d'un `struct`. Le non-respect de cette règle génère l'erreur de compilateur CS8156, « Impossible d'utiliser une expression dans ce contexte, car elle ne peut pas être retournée par référence ».

En outre, les valeurs de retour de référence ne sont pas autorisées sur les méthodes asynchrones. Une méthode asynchrone peut être retournée avant la fin de son exécution, même si sa valeur de retour est encore inconnue.

Définition d'une valeur de retour de référence

Une méthode qui retourne une *valeur de retour de référence* doit remplir les deux conditions suivantes :

- Dans la signature de méthode, le mot clé ref précède le type de retour.
- Pour chaque instruction return dans le corps de la méthode, le mot clé ref précède le nom de l'instance retournée.

L'exemple suivant montre une méthode qui remplit ces conditions et retourne une référence à un objet `Person` nommé `p` :

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Utilisation d'une valeur de retour de référence

La valeur de retour de référence est l'alias vers une autre variable dans l'étendue de la méthode appelée. Toute utilisation d'une valeur de retour de référence revient à utiliser la variable dont elle est l'alias :

- Quand vous lui affectez une valeur, vous affectez une valeur à la variable dont elle est l'alias.
- Quand vous lisez sa valeur, vous lisez la valeur de la variable dont elle est l'alias.
- Si vous la retournez *par référence*, vous retournez un alias vers cette même variable.
- Si vous la passez à une autre méthode *par référence*, vous passez une référence à la variable dont elle est l'alias.
- Quand vous créez un alias de variable locale ref, vous créez un nouvel alias vers la même variable.

Variables locales ref

Partez du principe que la méthode `GetContactInformation` est déclarée comme un retour de référence :

```
public ref Person GetContactInformation(string fname, string lname)
```

Une affectation par valeur lit la valeur d'une variable et l'affecte à une nouvelle variable :

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

L'affectation précédente déclare `p` comme une variable locale. Sa valeur initiale est copiée à partir de la lecture de la valeur retournée par `GetContactInformation`. Toute affectation future à `p` ne modifiera en rien la valeur de la variable renvoyée par `GetContactInformation`. La variable `p` n'est plus un alias vers la variable retournée.

Vous déclarez une variable *locale ref* pour copier l'alias vers la valeur d'origine. Dans l'affectation suivante, `p` est un alias vers la variable retournée à partir de `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

L'utilisation ultérieure de `p` revient à utiliser la variable retournée par `GetContactInformation`, car `p` est un alias de cette variable. Les modifications apportées à `p` modifient également la variable retournée à partir de `GetContactInformation`.

Le mot clé `ref` est utilisé à la fois avant la déclaration de la variable locale *et* avant l'appel de la méthode.

Vous pouvez accéder à une valeur par référence de la même façon. Dans certains cas, l'accès à une valeur par référence augmente les performances en évitant une opération de copie potentiellement coûteuse. Par exemple,

l'instruction suivante montre comment il est possible de définir une valeur locale ref qui est utilisée pour référencer une valeur.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

Le mot clé `ref` est utilisé à la fois avant la déclaration de la variable locale *et* avant la valeur du second exemple. Si les deux mots clés `ref` ne sont pas inclus dans la déclaration et l'affectation de la variable dans les deux exemples, l'erreur du compilateur CS8172, « Impossible d'initialiser une variable par référence avec une valeur » est générée.

Dans les versions antérieures à C# 7.3, les variables locales ref ne pouvaient pas être réassignées pour référencer un stockage différent après avoir été initialisées. Cette restriction a été supprimée. L'exemple suivant illustre une réassignation :

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Les variables locales ref doivent toujours être initialisées quand elles sont déclarées.

Retours ref et variables locales ref : exemple

L'exemple suivant définit une classe `NumberStore` qui stocke un tableau de valeurs entières. La méthode `FindNumber` retourne par référence le premier nombre supérieur ou égal au nombre passé comme argument. Si aucun nombre n'est supérieur ou égal à l'argument, la méthode retourne le nombre se trouvant à l'index 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

L'exemple suivant appelle la méthode `NumberStore.FindNumber` pour récupérer la première valeur supérieure ou égale à 16. L'appelant multiplie ensuite par deux la valeur retournée par la méthode. La sortie de l'exemple montre comment la modification est reflétée dans la valeur des éléments de tableau de l'instance `NumberStore`.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Si les valeurs de retour de référence ne sont pas prises en charge, une telle opération est effectuée en retournant l'index de l'élément de tableau ainsi que sa valeur de retour. L'appelant peut ensuite utiliser cet index pour modifier la valeur dans un appel de méthode distinct. Toutefois, l'appelant peut également modifier l'index pour accéder à d'autres valeurs de tableau et éventuellement les modifier.

L'exemple suivant montre comment réécrire la méthode `FindNumber` à compter de C# 7.3 pour utiliser la réassignation des variables locales ref :

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

Cette deuxième version est plus efficace avec des séquences plus longues dans les scénarios où le nombre recherché est proche de la fin du tableau.

Voir aussi

- ref, mot clé
- Écrire du code sécurisé et efficace

Passage de paramètres (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. Avec le passage par référence, les fonctions membres, les méthodes, les propriétés, les indexeurs, les opérateurs et les constructeurs peuvent changer la valeur des paramètres et rendre cette modification persistante dans l'environnement d'appel. Pour passer un paramètre par référence avec l'intention de changer la valeur, utilisez le mot clé `ref` ou `out`. Pour passer un paramètre par référence avec l'intention d'éviter la copie, mais de ne pas changer la valeur, utilisez le modificateur `in`. Pour plus de simplicité, les exemples de cette rubrique utilisent uniquement le mot clé `ref`. Pour plus d'informations sur la différence entre `in`, `ref` et `out`, consultez in, ref et out.

L'exemple suivant illustre la différence entre les paramètres de référence et de valeur.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

Pour plus d'informations, voir les rubriques suivantes :

- Passage de paramètres de type valeur
- Passage de paramètres de type référence

Spécification du langage C#

Pour plus d'informations, consultez Liste des arguments dans la Spécification du langage C#. La spécification du

langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Méthodes

Passage de paramètres de type valeur (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

Une variable de type valeur contient ses données directement, par opposition à une variable de type référence, qui contient une référence à ses données. Passer une variable de type valeur à une méthode par valeur signifie passer une copie de la variable à la méthode. Toutes les modifications apportées au paramètre qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur les données d'origine stockées dans la variable d'argument. Si vous souhaitez que la méthode appelée change la valeur de l'argument, vous devez la passer par référence, à l'aide du mot clé ref ou out. Vous pouvez également utiliser le mot clé in pour passer un paramètre de valeur par référence et éviter la copie tout en garantissant que la valeur n'est pas changée. Pour des raisons de simplicité, les exemples suivants utilisent `ref`.

Passage de types valeur par valeur

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passage de types valeur par référence

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Permutation de types valeur

L'un des exemples courants de changement de valeurs d'arguments est une méthode de permutation, où vous passez deux variables à la méthode et celle-ci permute leur contenu. Vous devez passer les arguments à la méthode de permutation par référence. Sinon, vous permutez des copies locales des paramètres à l'intérieur de la méthode, et aucune modification n'intervient dans la méthode d'appel. L'exemple suivant permute des valeurs d'entier.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Quand vous appelez la méthode `SwapByRef`, utilisez le mot-clé `ref` dans l'appel, comme illustré dans l'exemple suivant.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

Voir aussi

- Guide de programmation C#
- Passer des paramètres
- Passage de paramètres de type référence

Passage de paramètres de type référence (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Une variable d'un type référence ne contient pas directement ses données ; il contient une référence à ses données. Quand vous passez un paramètre de type référence par valeur, il est possible de changer les données appartenant à l'objet référencé, comme la valeur d'un membre de classe. En revanche, vous ne pouvez pas changer la valeur de la référence elle-même ; par exemple, vous ne pouvez pas utiliser la même référence pour allouer de la mémoire à un nouvel objet et la faire persister en dehors de la méthode. Pour cela, passez le paramètre en utilisant le mot clé ref ou out. Pour des raisons de simplicité, les exemples suivants utilisent `ref`.

Passage de types référence par valeur

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr
[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passage de types référence par référence

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Permutation de deux chaînes

La permutation de chaînes est un bon exemple de passage de paramètres de type référence par référence. Dans l'exemple, deux chaînes, `str1` et `str2`, sont initialisées dans `Main` et passées à la méthode `SwapStrings` en tant que paramètres modifiés par le mot clé `ref`. Les deux chaînes sont permutées à l'intérieur de la méthode mais aussi à l'intérieur de `Main`.

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);   // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/
```

Dans cet exemple, les paramètres doivent être passés par référence pour affecter les variables dans le programme appelant. Si vous supprimez le mot clé `ref` de l'en-tête et de l'appel de la méthode, aucune modification ne se produit dans le programme appelant.

Pour plus d'informations sur les chaînes, consultez string.

Voir aussi

- Guide de programmation C#
- Passer des paramètres
- ref
- in
- à
- Types référence

Comment connaître la différence entre le passage d'un struct et le passage d'une référence de classe à une méthode (Guide de programmation C#)

16/12/2020 • 4 minutes to read • Edit Online

L'exemple suivant montre comment le passage d'un struct à une méthode diffère du passage d'une instance de classe à une méthode. Dans l'exemple, les deux arguments (struct et instance de classe) sont passés par valeur, tandis que les deux méthodes changent la valeur d'un champ de l'argument. Toutefois, les résultats des deux méthodes ne sont pas identiques, car ce qui est passé quand vous passez un struct diffère de ce qui est passé quand vous passez une instance d'une classe.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

Exemple

```
class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Voir aussi

- Guide de programmation C#
- Classes
- Types de structure
- Passage de paramètres

Variables locales implicitement typées (Guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

Les variables locales peuvent être déclarées sans donner de type explicite. Le mot clé `var` indique au compilateur de déduire le type de la variable à partir de l'expression située à droite de l'instruction d'initialisation. Le type inféré peut être un type intégré, un type anonyme, un type défini par l'utilisateur ou un type défini dans la bibliothèque de classes .NET. Pour plus d'informations sur l'initialisation des tableaux avec `var`, consultez Tableaux implicitement typés.

Les exemples suivants montrent différentes manières de déclarer des variables locales avec `var` :

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

Il est important de comprendre que le mot clé `var` n'est pas « variant » et qu'il n'indique pas que la variable est peu typée ou à liaison tardive. Cela signifie simplement que le compilateur détermine et assigne le type qui convient le mieux.

Le mot clé `var` peut être utilisé dans les contextes suivants :

- Dans des variables locales (variables déclarées dans la portée de la méthode), comme indiqué dans l'exemple précédent.
- Dans une instruction d'initialisation for

  ```
  for (var x = 1; x < 10; x++)
  ```

- Dans une instruction d'initialisation foreach

  ```
  foreach (var item in list) {...}
  ```

- Dans une instruction using

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Pour plus d'informations, consultez comment utiliser des variables locales et des tableaux implicitement typés dans une expression de requête.

Types var et anonymes

Dans de nombreux cas, l'utilisation de `var` est facultative et sert uniquement à simplifier la syntaxe. Toutefois, lorsqu'une variable est initialisée avec un type anonyme, vous devez déclarer la variable en tant que `var` si vous savez déjà que vous aurez besoin d'accéder aux propriétés de l'objet. Il s'agit d'un scénario courant dans les expressions de requête LINQ. Pour plus d'informations, consultez types anonymes.

Du point de vue de votre code source, un type anonyme n'a pas de nom. Par conséquent, si une variable de requête a été initialisée avec `var`, la seule façon d'accéder aux propriétés de la séquence d'objets retournée consiste à utiliser `var` comme type pour la variable d'itération de l'instruction `foreach`.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Notes

Les restrictions suivantes s'appliquent aux déclarations de variables implicitement typées :

- `var` peut être utilisé uniquement lorsqu'une variable locale est déclarée et initialisée dans la même instruction. La variable ne peut pas être initialisée vers la valeur Null, vers un groupe de méthodes ou vers une fonction anonyme.
- `var` ne peut pas être utilisé dans les champs situés dans la portée de la classe.
- Les variables déclarées à l'aide de `var` ne peuvent pas être utilisées dans l'expression d'initialisation. En d'autres termes, cette expression est légale : `int i = (i = 20);` mais cette expression génère une erreur au moment de la compilation : `var i = (i = 20);`
- Il n'est pas possible d'initialiser plusieurs variables implicitement typées dans la même instruction.
- Si un type nommé `var` se trouve dans la portée, le mot clé `var` est résolu en ce nom de type et n'est

pas considéré comme faisant partie d'une déclaration de variable locale implicitement typée.

Le typage implicite avec le mot clé `var` ne peut être appliqué qu'aux variables comprises dans la portée de la méthode locale. Le typage implicite n'est pas disponible pour les champs de classe, car le compilateur C# rencontrerait un paradoxe logique pendant le traitement du code : le compilateur a besoin de connaître le type du champ, mais il ne peut pas déterminer le type tant que l'expression d'assignation n'est pas analysée, et l'expression ne peut pas être évaluée sans connaître le type. Considérez le code suivant :

```
private var bookTitles;
```

`bookTitles` est un champ de classe de type `var`. Le champ n'ayant aucune expression à évaluer, le compilateur ne peut pas déduire le type que `bookTitles` est censé avoir. De plus, l'ajout d'une expression au champ (comme vous le feriez pour une variable locale) est également insuffisant :

```
private var bookTitles = new List<string>();
```

Quand le compilateur rencontre des champs pendant la compilation du code, il enregistre le type de chaque champ avant de traiter toutes les expressions associées. Le compilateur rencontre le même paradoxe en essayant d'analyser `bookTitles` : il doit connaître le type du champ, mais il détermine normalement le type de `var` en analysant l'expression, ce qui est impossible sans connaître le type au préalable.

`var` peut également être utile avec les expressions de requête dans lesquelles il est difficile de déterminer le type construit exact de la variable de requête. Cela peut se produire avec les opérations de regroupement et de classement.

Le mot clé `var` peut également être utile lorsque le type de la variable est fastidieux à taper, ou bien lorsqu'il est évident ou lorsqu'il n'aide pas à la lisibilité du code. Par exemple, `var` est utile avec les types génériques imbriqués tels que ceux utilisés avec les opérations de groupe. Dans la requête suivante, le type de la variable de requête est `IEnumerable<IGrouping<string, Student>>`. Tant que les personnes qui doivent gérer votre code comprennent ceci, le typage implicite peut tout à fait être utilisé pour rendre votre code plus concis et simplifier sa gestion.

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

L'utilisation de `var` permet de simplifier votre code, mais son utilisation doit être limitée aux cas où il est requis ou lorsqu'il rend votre code plus facile à lire. Pour plus d'informations sur l'utilisation `var` correcte de, consultez la section variables locales implicitement typées dans l'article instructions de codage C#.

Voir aussi

- Référence C#
- Tableaux implicitement typés
- Comment : utiliser des tableaux et des variables locales implicitement typés dans une expression de requête
- Types anonymes
- Initialiseurs d'objets et de collections
- var
- LINQ en C#

- LINQ (Language Integrated Query)
- for
- foreach, in
- using, instruction

Comment utiliser des variables locales et des tableaux implicitement typés dans une expression de requête (Guide de programmation C#)

16/12/2020 • 3 minutes to read • Edit Online

Vous pouvez utiliser des variables locales implicitement typées chaque fois que vous voulez que le compilateur détermine le type d'une variable locale. Vous devez utiliser des variables locales implicitement typées pour stocker des types anonymes, qui sont souvent utilisés dans les expressions de requête. Les exemples suivants illustrent des utilisations facultatives et obligatoires de variables locales implicitement typées dans des requêtes.

Les variables locales implicitement typées sont déclarées à l'aide du mot clé contextuel var. Pour plus d'informations, consultez Variables locales implicitement typées et Tableaux implicitement typés.

Exemple

L'exemple suivant illustre un scénario courant dans lequel le mot clé `var` est obligatoire : une expression de requête qui produit une séquence de types anonymes. Dans ce scénario, la variable de requête et la variable d'itération figurant dans l'instruction `foreach` doivent être implicitement typées avec `var`, car vous n'avez pas accès à un nom de type pour le type anonyme. Pour plus d'informations sur les types anonymes, consultez Types anonymes.

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

Exemple

Dans l'exemple suivant, le mot clé `var` est utilisé dans un cas similaire, à la différence que l'utilisation de `var` est facultative. Comme `student.LastName` est une chaîne, l'exécution de la requête retourne une séquence de chaînes. De ce fait, le type de `queryID` pourrait être déclaré en tant que `System.Collections.Generic.IEnumerable<string>` au lieu de `var`. Le mot clé `var` est utilisé pour des raisons pratiques. Dans l'exemple, la variable d'itération figurant dans l'instruction `foreach` est explicitement typée en tant que chaîne, mais elle aurait pu être déclarée à l'aide de `var`. Comme le type de la variable d'itération n'est pas un type anonyme, l'utilisation de `var` est facultative et non obligatoire. Pour rappel, `var` n'est pas en soi un type, mais une instruction indiquant au compilateur de déduire et d'assigner le type.

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

Voir aussi

- Guide de programmation C#
- Méthodes d'extension
- LINQ (Language Integrated Query)
- var
- LINQ en C#

Méthodes d'extension (Guide de programmation C#)

02/11/2020 • 17 minutes to read • Edit Online

Les méthodes d'extension vous permettent d'« ajouter » des méthodes à des types existants sans créer un type dérivé, ni recompiler ou modifier le type d'origine. Les méthodes d'extension sont des méthodes statiques, mais elles sont appelées comme s'il s'agissait de méthodes d'instance sur le type étendu. Pour le code client écrit en C#, F # et Visual Basic, il n'y a aucune différence apparente entre appeler une méthode d'extension et les méthodes définies dans un type.

Les méthodes d'extension les plus courantes sont les opérateurs de requête standard LINQ qui ajoutent des fonctionnalités de requête aux System.Collections.IEnumerable types et existants System.Collections.Generic.IEnumerable<T> . Pour utiliser les opérateurs de requête standard, introduisez-les d'abord dans la portée avec une directive `using System.Linq`. Puis, tout type qui implémente IEnumerable<T> semble avoir des méthodes d'instance telles que GroupBy, OrderBy, Average, etc. Vous pouvez consulter ces méthodes supplémentaires dans la saisie semi-automatique des instructions IntelliSense quand vous tapez un « point » après une instance d'un type IEnumerable<T> tel que List<T> ou Array.

Exemple OrderBy

L'exemple suivant indique comment appeler la méthode `OrderBy` d'opérateur de requête standard sur un tableau d'entiers. L'expression entre parenthèses est une expression lambda. De nombreux opérateurs de requête standard prennent des expressions lambda comme paramètres, mais ce n'est pas une exigence pour les méthodes d'extension. Pour plus d'informations, consultez expressions lambda.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Les méthodes d'extension sont définies comme méthodes statiques mais sont appelées en utilisant la syntaxe de méthode d'instance. Leur premier paramètre spécifie le type sur lequel la méthode s'exécute. Le paramètre est précédé du modificateur This . Les méthodes d'extension sont uniquement dans la portée lorsque vous importez explicitement l'espace de noms dans votre code source avec une directive `using`.

L'exemple suivant présente une méthode d'extension définie pour la classe System.String. Elle est définie à l'intérieur d'une classe statique non imbriquée et non générique :

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

La méthode d'extension `WordCount` peut être mise à portée avec cette directive `using` :

```
using ExtensionMethods;
```

Elle peut être appelée à partir d'une application à l'aide de cette syntaxe :

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Vous appelez la méthode d'extension dans votre code avec la syntaxe de méthode d'instance. Le langage intermédiaire (IL) généré par le compilateur traduit votre code en un appel sur la méthode statique. Le principe de l'encapsulation n'est pas réellement violé. Les méthodes d'extension ne peuvent pas accéder aux variables privées dans le type qu'elles étendent.

Pour plus d'informations, consultez comment implémenter et appeler une méthode d'extension personnalisée.

En général, vous appellerez probablement les méthodes d'extension beaucoup plus souvent que d'implémenter votre propre méthode. Comme les méthodes d'extension sont appelées à l'aide de la syntaxe de méthode d'instance, aucune connaissance particulière n'est requise pour les utiliser depuis le code client. Pour activer des méthodes d'extension pour un type particulier, ajoutez simplement une directive `using` pour l'espace de noms dans lequel les méthodes sont définies. Par exemple, pour utiliser les opérateurs de requête standard, ajoutez la directive `using` à votre code :

```
using System.Linq;
```

(Vous devrez peut-être également ajouter une référence à System.Core.dll.) Vous remarquerez que les opérateurs de requête standard apparaissent désormais dans IntelliSense comme des méthodes supplémentaires disponibles pour la plupart des IEnumerable<T> types.

Liaison de méthodes d'extension à la compilation

Vous pouvez utiliser des méthodes d'extension pour étendre une classe ou une interface, mais pas pour les remplacer. Une méthode d'extension avec le même nom et la même signature qu'une méthode d'interface ou de classe ne sera jamais appelée. Au moment de la compilation, les méthodes d'extension ont toujours la priorité la plus faible par rapport aux méthodes d'instance définies dans le type lui-même. En d'autres termes, si un type a une méthode nommée `Process(int i)` et que vous avez une méthode d'extension avec la même signature, le compilateur créera toujours une liaison avec la méthode d'instance. Lorsque le compilateur rencontre un appel de méthode, il recherche d'abord une correspondance dans les méthodes d'instance du type. Si aucune correspondance n'est trouvée, il recherche toutes les méthodes d'extension définies pour le type et crée une liaison avec la première méthode d'extension qu'il trouve. L'exemple

suivant montre comment le compilateur détermine quelle méthode d'extension ou méthode d'instance est choisie pour créer une liaison.

Exemple

L'exemple suivant montre les règles que le compilateur C# suit pour déterminer s'il faut lier un appel de méthode à une méthode d'instance sur le type, ou à une méthode d'extension. Le classe statique `Extensions` contient des méthodes d'extension définies pour tout type qui implémente `IMyInterface`. Les classes `A`, `B` et `C` implémentent toutes l'interface.

La méthode d'extension `MethodB` n'est jamais appelée car son nom et sa signature correspondent exactement aux méthodes déjà implémentées par les classes.

Lorsque le compilateur ne trouve pas de méthode d'instance avec une signature correspondante, il crée une liaison avec une méthode d'extension correspondante, s'il en existe une.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
```

```
namespace ExtensionMethodDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
```

```
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Modèles d'utilisation courants

Fonctionnalités des collections

Dans le passé, il était courant de créer des « classes de collection » implémentant l' System.Collections.Generic.IEnumerable<T> interface pour un type donné et des fonctionnalités contenues ayant agi sur des collections de ce type. Bien qu'il n'y ait rien de mal à créer ce type d'objet de collection, la même fonctionnalité peut être obtenue à l'aide d'une extension sur le System.Collections.Generic.IEnumerable<T> . Les extensions présentent l'avantage de permettre l'appel de la fonctionnalité à partir de n'importe quelle collection telle que System.Array ou System.Collections.Generic.List<T> qui implémente System.Collections.Generic.IEnumerable<T> sur ce type. Vous trouverez un exemple d'utilisation d'un tableau de Int32 plus haut dans cet article.

Fonctionnalités spécifiques aux couches

Lorsque vous utilisez une architecture d'oignon ou une autre conception d'application en couches, il est courant d'avoir un ensemble d'entités de domaine ou de Transfert de données objets qui peuvent être utilisés pour communiquer à travers les limites de l'application. Ces objets ne contiennent généralement aucune fonctionnalité, ou uniquement des fonctionnalités minimales qui s'appliquent à toutes les couches de l'application. Les méthodes d'extension peuvent être utilisées pour ajouter des fonctionnalités spécifiques à chaque couche d'application sans avoir à charger l'objet avec des méthodes qui ne sont pas nécessaires ou qui veulent dans d'autres couches.

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Extension des types prédéfinis

Plutôt que de créer des objets quand des fonctionnalités réutilisables doivent être créées, nous pouvons souvent étendre un type existant, tel qu'un type .NET ou CLR. Par exemple, si nous n'utilisons pas les méthodes d'extension, nous pouvons créer une `Engine` `Query` classe ou pour effectuer le travail d'exécution d'une requête sur un SQL Server qui peut être appelé à partir de plusieurs emplacements dans notre code. Toutefois, nous pouvons étendre la System.Data.SqlClient.SqlConnection classe à l'aide de méthodes d'extension pour effectuer cette requête depuis n'importe quel endroit où nous disposons d'une connexion à un SQL Server. D'autres exemples peuvent être d'ajouter des fonctionnalités communes à la System.String classe, d'étendre les fonctionnalités de traitement des données des System.IO.File System.IO.Stream objets et, ainsi que des System.Exception objets pour des fonctionnalités de gestion des erreurs spécifiques. Ces types de cas d'usage sont limités uniquement par votre imagination et bon sens.

L'extension de types prédéfinis peut être difficile avec les `struct` types, car ils sont passés par valeur aux

méthodes. Cela signifie que toute modification apportée au struct est apportée à une copie du struct. Ces modifications ne sont pas visibles une fois que la méthode d'extension se termine. À compter de C# 7,2, vous pouvez ajouter le `ref` modificateur au premier argument d'une méthode d'extension. L'ajout du `ref` modificateur signifie que le premier argument est passé par référence. Cela vous permet d'écrire des méthodes d'extension qui modifient l'état du struct qui est étendu.

Instructions générales

Bien qu'il soit toujours considéré comme préférable d'ajouter des fonctionnalités en modifiant le code d'un objet ou en dérivant un nouveau type chaque fois qu'il est raisonnable et possible de le faire, les méthodes d'extension sont devenues une option cruciale pour créer des fonctionnalités réutilisables dans l'écosystème .NET. Dans les cas où la source d'origine n'est pas sous votre contrôle, lorsqu'un objet dérivé est inapproprié ou impossible, ou lorsque les fonctionnalités ne doivent pas être exposées au-delà de la portée applicable, les méthodes d'extension sont un excellent choix.

Pour plus d'informations sur les types dérivés, consultez héritage.

Lors de l'utilisation d'une méthode d'extension pour étendre un type dont le code source n'est pas contrôle, vous courez le risque qu'une modification de l'implémentation du type entraînera l'arrêt de votre méthode d'extension.

Si vous implémentez des méthodes d'extension pour un type donné, prenez en compte les points suivants :

- Une méthode d'extension ne sera jamais appelée si elle a la même signature qu'une méthode définie dans le type.
- Les méthodes d'extension sont mises en portée au niveau de l'espace de noms. Par exemple, si vous avez plusieurs classes statiques qui contiennent des méthodes d'extension dans un espace de noms unique nommé `Extensions` , elles seront toutes mises en portée par la `using Extensions;` directive.

Pour une bibliothèque de classes que vous avez implémentée, vous ne devez pas utiliser de méthodes d'extension pour éviter d'incrémenter le numéro de version d'un assembly. Si vous souhaitez ajouter des fonctionnalités importantes à une bibliothèque pour laquelle vous êtes propriétaire du code source, suivez les instructions .NET pour le contrôle de version des assemblys. Pour plus d'informations, consultez Versioning des assemblys.

Voir aussi

- Guide de programmation C#
- Exemples de programmation parallèle (il s'agit de nombreux exemples de méthodes d'extension)
- Expressions lambda
- Vue d'ensemble des opérateurs de requête standard
- Règles de conversion pour les paramètres Instance et leur impact
- Interopérabilité des méthodes d'extension entre les langages
- Méthodes d'extension et délégués curryfiés
- Extension method Binding and Error reporting

Comment implémenter et appeler une méthode d'extension personnalisée (Guide de programmation C#)

16/12/2020 • 3 minutes to read • Edit Online

Cette rubrique montre comment implémenter vos propres méthodes d'extension pour n'importe quel type .NET. Le code client peut utiliser vos méthodes d'extension en ajoutant une référence à la DLL qui les contient, et en ajoutant une directive using qui spécifie l'espace de noms dans lequel les méthodes d'extension sont définies.

Pour définir et appeler la méthode d'extension

1. Définissez une classe statiquepour contenir la méthode d'extension.

   La classe doit être visible par le code client. Pour plus d'informations sur les règles d'accessibilité, consultez Modificateurs d'accès.

2. Implémentez la méthode d'extension en tant que méthode statique avec au moins la même visibilité que la classe conteneur.

3. Le premier paramètre de la méthode spécifie le type sur lequel la méthode opère. Il doit être précédé du modificateur this.

4. Dans le code appelant, ajoutez une directive `using` pour spécifier l'espace de noms qui contient la classe de méthode d'extension.

5. Appelez les méthodes comme s'il s'agissait de méthodes d'instance sur le type.

   Notez que le premier paramètre n'est pas spécifié par le code appelant, car il représente le type sur lequel l'opérateur est appliqué, et le compilateur connaît déjà le type de votre objet. Vous devez fournir des arguments uniquement pour les paramètres 2 à `n`.

Exemple

L'exemple suivant implémente une méthode d'extension nommée `WordCount` dans la classe `CustomExtensions.StringExtension`. La méthode opère sur la classe String, qui est spécifiée comme premier paramètre de méthode. L'espace de noms `CustomExtensions` est importé dans l'espace de noms d'application, et la méthode est appelée à l'intérieur de la méthode `Main`.

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

Sécurité .NET

Les méthodes d'extension ne présentent aucune faille de sécurité spécifique. Elles ne peuvent jamais être utilisées pour emprunter l'identité des méthodes existantes sur un type, car toutes les collisions de noms sont résolues en faveur de l'instance ou de la méthode statique définie par le type lui-même. Les méthodes d'extension ne peuvent pas accéder à des données privées dans la classe étendue.

Voir aussi

- Guide de programmation C#
- Méthodes d'extension
- LINQ (Language Integrated Query)
- Classes statiques et membres de classe statique
- protected
- intérieurs
- public
- this
- namespace

Comment créer une nouvelle méthode pour une énumération (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Vous pouvez utiliser des méthodes d'extension pour ajouter des fonctionnalités propres à un type enum particulier.

Exemple

Dans l'exemple suivant, l'énumération `Grades` représente les notes qu'un étudiant peut obtenir dans une classe. Une méthode d'extension nommée `Passing` est ajoutée au type `Grades` pour que chaque instance de ce type « sache » maintenant si elle représente une note au-dessus de la moyenne.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Notez que la classe `Extensions` contient également une variable statique qui est mise à jour de manière

dynamique, et que la valeur de retour de la méthode d'extension reflète la valeur actuelle de cette variable. Cela démontre que, dans les coulisses, les méthodes d'extension sont appelées directement sur la classe statique dans laquelle elles sont définies.

Voir aussi

- Guide de programmation C#
- Méthodes d'extension

Arguments nommés et facultatifs (Guide de programmation C#)

16/12/2020 • 13 minutes to read • Edit Online

C# 4 introduit des arguments nommés et facultatifs. Les *arguments nommés* vous permettent de spécifier un argument pour un paramètre en faisant correspondre l'argument avec son nom plutôt qu'avec sa position dans la liste de paramètres. Les *arguments facultatifs* vous permettent d'omettre des arguments pour certains paramètres. Les deux techniques peuvent être utilisées avec les méthodes, les indexeurs, les constructeurs et les délégués.

Quand vous utilisez des arguments nommés et facultatifs, ils sont évalués selon leur ordre d'affichage dans la liste d'arguments, et non dans la liste de paramètres.

Les paramètres nommés et facultatifs vous permettent de fournir des arguments pour les paramètres sélectionnés. Cette fonctionnalité simplifie les appels aux interfaces COM, telles que les API Microsoft Office Automation.

Arguments nommés

Les arguments nommés vous libèrent de l'ordre des paramètres dans les listes de paramètres des méthodes appelées. Vous pouvez spécifier le paramètre de chaque argument par son nom. Par exemple, une fonction qui imprime les détails d'une commande (par exemple, le nom du vendeur, le numéro de commande & nom du produit) peut être appelée en envoyant des arguments par position, dans l'ordre défini par la fonction.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Si vous ne vous souvenez pas de l'ordre des paramètres, mais que vous connaissez leur nom, vous pouvez envoyer les arguments dans n'importe quel ordre.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Les arguments nommés améliorent également la lisibilité de votre code en identifiant ce que chaque argument représente. Dans l'exemple de méthode ci-dessous, l' `sellerName` ne peut pas être null ou un espace blanc. `sellerName` et `productName` étant tous deux des types de chaînes, au lieu d'envoyer les arguments par position, il est plus logique d'utiliser des arguments nommés pour lever l'ambiguïté entre les deux et réduire les risques de confusion pour toute personne lisant le code.

Les arguments nommés, quand ils sont utilisés avec des arguments de position, sont valides tant que :

- Ils ne sont pas suivis d'arguments de position, ou

  ```
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- *À compter de C# 7.2*, ils sont utilisés dans la position correcte Dans l'exemple ci-dessous, le paramètre `orderNum` est à la position correcte, mais il n'est pas explicitement nommé.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Les arguments positionnels qui suivent les arguments nommés non ordonnés ne sont pas valides.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Exemple

Le code suivant implémente les exemples de cette section, ainsi que d'autres exemples.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Arguments facultatifs

La définition d'une méthode, d'un constructeur, d'un indexeur ou d'un délégué peut spécifier que ses paramètres sont obligatoires ou facultatifs. Chaque appel doit fournir des arguments pour tous les paramètres obligatoires, mais peut omettre les arguments des paramètres facultatifs.

Dans sa définition, chaque paramètre facultatif a une valeur par défaut. Si aucun argument n'est envoyé pour ce paramètre, la valeur par défaut est utilisée. Une valeur par défaut doit être l'un des types d'expressions suivants :

- une expression constante ;
- une expression de la forme `new ValType()`, où `ValType` est un type valeur (par exemple, enum ou struct) ;

- une expression de la forme default(ValType), où `ValType` est un type valeur.

Les paramètres facultatifs sont définis à la fin de la liste de paramètres, après tous les paramètres obligatoires. Si l'appelant fournit un argument pour l'un des paramètres d'une série de paramètres facultatifs, il doit fournir des arguments pour tous les paramètres facultatifs précédents. Les écarts séparés par des virgules dans la liste d'arguments ne sont pas pris en charge. Par exemple, dans le code suivant, la méthode d'instance `ExampleMethod` est définie avec un paramètre obligatoire et deux paramètres facultatifs.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

L'appel suivant à `ExampleMethod` provoque une erreur du compilateur, car un argument est fourni pour le troisième paramètre, mais pas pour le deuxième.

```
//anExample.ExampleMethod(3, ,4);
```

Toutefois, si vous connaissez le nom du troisième paramètre, vous pouvez utiliser un argument nommé pour effectuer la tâche.

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense indique les paramètres facultatifs par des crochets, comme illustré dans l'exemple suivant :

```
anExample.ExampleMethod(
  void ExampleClass.ExampleMethod(int required,
                                  [string optionalstr = "default string"],
                                  [int optionalint = 10])
```

> **NOTE**
>
> Vous pouvez aussi déclarer des paramètres facultatifs à l'aide de la classe OptionalAttribute .NET. Les paramètres `OptionalAttribute` ne nécessitent pas de valeur par défaut.

Exemple

Dans l'exemple suivant, le constructeur associé à `ExampleClass` a un seul paramètre, qui est facultatif. La méthode d'instance `ExampleMethod` a un paramètre obligatoire (`required`) et deux paramètres facultatifs (`optionalstr` et `optionalint`). Le code dans `Main` montre les différentes façons dont le constructeur et la méthode peuvent être appelés.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
```

```
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

Le code précédent illustre un certain nombre d'exemples où les paramètres facultatifs ne sont pas appliqués correctement. Le premier illustre qu'un argument doit être fourni pour le premier paramètre, ce qui est obligatoire.

Interfaces COM

Les arguments nommés et facultatifs, ainsi que la prise en charge des objets dynamiques, améliorent beaucoup l'interopérabilité avec les API COM, telles que les API d'automatisation d'Office.

Par exemple, la méthode AutoFormat de l'interface Range de Microsoft Office Excel comporte sept paramètres, tous facultatifs. Ces paramètres sont indiqués dans l'illustration suivante :

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

Dans C# 3.0 et les versions antérieures, un argument est obligatoire pour chaque paramètre, comme indiqué dans l'exemple suivant.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

Toutefois, vous pouvez considérablement simplifier l'appel à `AutoFormat` en utilisant les arguments nommés et facultatifs introduits dans C# 4.0. Les arguments nommés et facultatifs vous permettent d'omettre l'argument d'un paramètre facultatif si vous ne souhaitez pas modifier la valeur par défaut du paramètre. Dans l'appel suivant, une valeur est spécifiée pour un seul des sept paramètres.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

Pour plus d'informations et d'exemples, consultez comment utiliser des arguments nommés et facultatifs dans la programmation Office et Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#.

Résolution de surcharge

L'utilisation d'arguments nommés et facultatifs affecte la résolution de surcharge des manières suivantes :

- Une méthode, un indexeur ou un constructeur est un candidat pour l'exécution si chacun de ses paramètres est facultatif ou correspond, par nom ou par position, à un seul argument dans l'instruction appelante, et que cet argument peut être converti vers le type du paramètre.
- Si plusieurs candidats sont trouvés, les règles de résolution de surcharge des conversions préférées sont appliquées aux arguments qui sont explicitement spécifiés. Les arguments omis pour les paramètres facultatifs sont ignorés.
- Si deux candidats sont jugés corrects, la préférence passe à un candidat qui n'a pas de paramètres facultatifs pour lesquels des arguments ont été omis dans l'appel. La résolution de surcharge préfère généralement les candidats qui ont moins de paramètres.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Comment utiliser des arguments nommés et facultatifs dans la programmation Office (Guide de programmation C#)

16/12/2020 • 8 minutes to read • Edit Online

Les arguments nommés et les arguments facultatifs, qui ont été introduits avec C#4, rendent la programmation en C# plus pratique, plus souple et plus lisible. De plus, ces fonctionnalités facilitent considérablement l'accès aux interfaces COM, telles que les API Microsoft Office Automation.

Dans l'exemple suivant, la méthode ConvertToTable a seize paramètres qui représentent les caractéristiques d'une table, comme le nombre de colonnes et de lignes, la mise en forme, les bordures, les polices et les couleurs. Ces seize paramètres sont facultatifs, car la plupart du temps, il n'est pas nécessaire de spécifier des valeurs pour chacun d'eux. Toutefois, en l'absence d'arguments nommés et facultatifs, une valeur ou une valeur d'espace réservé doit être fournie pour chaque paramètre. Avec les arguments nommés et facultatifs, vous spécifiez des valeurs uniquement pour les paramètres qui sont nécessaires à votre projet.

Pour que vous puissiez effectuer ces procédures, Microsoft Office Word doit être installé sur votre ordinateur.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Pour créer une application console

1. Démarrez Visual Studio.
2. Dans le menu **Fichier** , pointez sur **Nouveau**, puis cliquez sur **Projet**.
3. Dans le volet **Catégories de modèles**, développez **Visual C#**, puis cliquez sur **Windows**.
4. Regardez en haut du volet **Modèles** pour vérifier que **.NET Framework 4**, ou version ultérieure, apparaît dans la zone **Framework cible**.
5. Dans le volet **Modèles**, cliquez sur **Application console**.
6. Attribuez un nom à votre projet dans le champ **Nom**.
7. Cliquez sur **OK**.

   Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Pour ajouter une référence

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis cliquez sur **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche.
2. Dans la page **.NET**, sélectionnez **Microsoft.Office.Interop.Word** dans la liste **Nom du composant**.
3. Cliquez sur **OK**.

Pour ajouter les directives using nécessaires

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier *Program.cs*, puis cliquez sur **Afficher le code**.

2. Ajoutez les `using` directives suivantes en haut du fichier de code :

```
using Word = Microsoft.Office.Interop.Word;
```

Pour afficher du texte dans un document Word

1. Dans la `Program` classe de *Program.cs*, ajoutez la méthode suivante pour créer une application Word et un document Word. La méthode Add comprend quatre paramètres facultatifs. Cet exemple utilise leurs valeurs par défaut. Par conséquent, aucun argument n'est nécessaire dans l'instruction appelante.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Ajoutez le code suivant à la fin de la méthode pour définir l'emplacement d'affichage du texte dans le document et le texte à afficher :

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Pour exécuter l'application

1. Ajoutez l'instruction suivante à main :

```
DisplayInWord();
```

2. Appuyez sur `CTRL + F5` pour exécuter le projet. Un document Word contenant le texte spécifié s'affiche.

Pour convertir le texte en tableau

1. Utilisez la méthode `ConvertToTable` pour inclure le texte dans un tableau. La méthode comprend seize paramètres facultatifs. IntelliSense place les paramètres facultatifs entre crochets, comme illustré dans l'exemple suivant.

```
range.ConvertToTable(
```

```
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Les arguments nommés et facultatifs permettent de spécifier des valeurs uniquement pour les paramètres que vous voulez modifier. Ajoutez le code suivant à la fin de la méthode `DisplayInWord` pour créer un tableau simple. L'argument spécifie que les virgules présentes dans le texte de la chaîne comprise dans `range` séparent les cellules du tableau.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Dans les versions antérieures de C#, l'appel à `ConvertToTable` requiert un argument de référence pour chaque paramètre, comme indiqué dans le code suivant :

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Appuyez sur `CTRL + F5` pour exécuter le projet.

Pour tester d'autres paramètres

1. Pour modifier la table afin qu'elle ait une colonne et trois lignes, remplacez la dernière ligne de `DisplayInWord` par l'instruction suivante, puis tapez `CTRL + F5`.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. Pour spécifier un format prédéfini pour la table, remplacez la dernière ligne de `DisplayInWord` par l'instruction suivante, puis tapez `CTRL + F5`. Le format peut être n'importe quelle constante WdTableFormat.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Exemple

Le code suivant contient l'exemple complet :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

Voir aussi

- Arguments nommés et facultatifs

Constructeurs (guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Chaque fois qu'une classe ou un struct est créé, son constructeur est appelé. Une classe ou un struct peut avoir plusieurs constructeurs qui prennent des arguments différents. Les constructeurs permettent au programmeur de définir des valeurs par défaut, de limiter l'instanciation et d'écrire un code flexible et facile à lire. Pour obtenir plus d'informations et d'exemples, consultez Utilisation de constructeurs et Constructeurs d'instances.

Constructeurs sans paramètre

Si vous ne fournissez pas de constructeur pour votre classe, C# en crée un par défaut qui instancie l'objet et définit des variables membres avec les valeurs par défaut, comme indiqué dans l'article valeurs par défaut des types C# . Si vous ne fournissez pas de constructeur pour votre struct, C# s'appuie sur un *constructeur sans paramètre implicite* pour initialiser automatiquement chaque champ à sa valeur par défaut. Pour plus d'informations et d'exemples, consultez constructeurs d'instances.

Syntaxe du constructeur

Un constructeur est une méthode dont le nom est le même que celui de son type. Sa signature de méthode inclut uniquement le nom de la méthode et sa liste de paramètres ; elle n'inclut pas de type de retour. L'exemple suivant affiche le constructeur d'une classe nommée `Person`.

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Si un constructeur peut être implémenté comme une seule instruction, vous pouvez utiliser une définition de corps d'expression. L'exemple suivant définit une classe `Location` dont le constructeur a un seul paramètre de chaîne nommé *name*. La définition de corps d'expression assigne l'argument au champ `locationName`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Constructeurs statiques

Les exemples précédents ont tous montré des constructeurs d'instances qui permettent de créer un objet. Une classe ou un struct peut également avoir un constructeur statique qui initialise les membres statiques du type. Les constructeurs statiques sont sans paramètre. Si vous ne fournissez pas de constructeur statique pour initialiser les champs statiques, le compilateur C# initialise les champs statiques à leur valeur par défaut, comme indiqué dans l'article valeurs par défaut des types C# .

L'exemple suivant utilise un constructeur statique pour initialiser un champ statique.

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

Vous pouvez également définir un constructeur statique avec une définition de corps d'expression, comme l'illustre l'exemple suivant.

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

Pour obtenir plus d'informations et d'exemples, consultez Constructeurs statiques.

Dans cette section

Utilisation de constructeurs

Constructeurs d'instances

Constructeurs privés

Constructeurs statiques

Comment : écrire un constructeur de copie

Voir aussi

- Guide de programmation C#
- Classes et structs
- Finaliseurs

- static
- Pourquoi les initialiseurs s'exécutent-ils dans l'ordre inverse en tant que constructeurs ? Première partie

Utilisation de constructeurs (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Quand une classe ou un struct est créé, son constructeur est appelé. Les constructeurs portent le même nom que la classe ou le struct, et ils initialisent généralement les membres de données du nouvel objet.

Dans l'exemple suivant, une classe nommée `Taxi` est définie en utilisant un constructeur simple. Cette classe est ensuite instanciée à l'aide de l'opérateur new. Le constructeur `Taxi` est appelé par l'opérateur `new` immédiatement après l'allocation de la mémoire pour le nouvel objet.

```
public class Taxi
{
    public bool IsInitialized;
    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

Un constructeur qui ne prend pas de paramètres est appelé *constructeur sans paramètre*. Les constructeurs sans paramètre sont appelés chaque fois qu'un objet est instancié à l'aide de l'opérateur `new` et qu'aucun argument n'est fourni à `new`. Pour plus d'informations, consultez Constructeurs d'instances.

À moins d'être statiques, les classes sans constructeur se voient attribuer un constructeur public sans paramètre par le compilateur C#, afin d'activer l'instanciation de classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Vous pouvez empêcher qu'une classe soit instanciée en rendant le constructeur privé, comme suit :

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Pour plus d'informations, consultez Constructeurs privés.

Les constructeurs des types struct ressemblent aux constructeurs de classe, mais les `structs` ne peuvent pas contenir de constructeur explicite sans paramètre, car le compilateur en fournit automatiquement un. Ce constructeur initialise chaque champ du `struct` à la valeur par défaut. Toutefois, ce constructeur sans paramètre est appelé uniquement si le `struct` est instancié avec `new`. Par exemple, ce code utilise le constructeur sans paramètre pour Int32. Vous avez ainsi la garantie que l'entier est initialisé :

```
int i = new int();
Console.WriteLine(i);
```

Toutefois, le code suivant provoque une erreur de compilateur, car il n'utilise pas `new` et il essaie d'utiliser un objet qui n'a pas été initialisé :

```
int i;
Console.WriteLine(i);
```

Les objets basés sur des `structs` (notamment tous les types numériques intégrés) peuvent également être initialisés ou assignés, puis utilisés, comme dans l'exemple suivant :

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

L'appel du constructeur sans paramètre pour un type valeur n'est donc pas obligatoire.

Les classes et les `structs` peuvent tous les deux définir des constructeurs qui prennent des paramètres. Les constructeurs qui prennent des paramètres doivent être appelés à l'aide d'une instruction `new` ou d'une instruction base. Les classes et les `structs` peuvent également définir plusieurs constructeurs, et ni les classes ni les structs ne sont nécessaires pour définir un constructeur sans paramètre. Par exemple :

```
public class Employee
{
    public int Salary;

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

Cette classe peut être créée à l'aide de l'une ou l'autre des instructions suivantes :

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Un constructeur peut utiliser le mot clé `base` pour appeler le constructeur d'une classe de base. Par exemple :

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

Dans cet exemple, le constructeur de la classe de base est appelé avant que le bloc du constructeur ne soit exécuté. Le mot clé `base` peut être utilisé avec ou sans paramètres. Tous les paramètres du constructeur peuvent être utilisés comme paramètres pour `base` ou comme partie d'une expression. Pour plus d'informations, consultez base.

Dans une classe dérivée, si un constructeur de classe de base n'est pas appelé explicitement à l'aide du mot clé `base`, le constructeur sans paramètre, s'il existe, est appelé implicitement. Cela signifie que les déclarations de constructeur suivantes sont en fait les mêmes :

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Si une classe de base n'offre pas de constructeur sans paramètre, la classe dérivée doit faire un appel explicite à un constructeur de base à l'aide de `base`.

Un constructeur peut appeler un autre constructeur dans le même objet à l'aide du mot clé this. Comme `base`, `this` peut être utilisé avec ou sans paramètres, et tous les paramètres dans le constructeur sont disponibles comme paramètres pour `this` ou comme partie d'une expression. Par exemple, le deuxième constructeur de l'exemple précédent peut être récrit à l'aide de `this` :

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

L'utilisation du mot clé `this` dans l'exemple précédent provoque l'appel de ce constructeur :

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Les constructeurs peuvent être marqués comme public, private, protected, internal, protected internal ou private protected. Ces modificateurs d'accès définissent la façon dont les utilisateurs de la classe peuvent construire la classe. Pour plus d'informations, consultez Modificateurs d'accès.

Un constructeur peut être déclaré statique à l'aide du mot clé static. Les constructeurs statiques sont appelés automatiquement, juste avant que des champs statiques soient accessibles, et ils sont généralement utilisés pour initialiser des membres de classe statique. Pour plus d'informations, consultez Constructeurs statiques.

Spécification du langage C#

Pour plus d'informations, consultez Constructeurs d'instances et Constructeurs statiques dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Constructeurs
- Finaliseurs

Constructeurs d'instances (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Les constructeurs d'instances sont utilisés pour créer et initialiser toutes les variables membres d'instance quand vous utilisez l'expression new pour créer un objet d'une classe. Pour initialiser une classe statique ou des variables statiques dans une classe non statique, vous devez définir un constructeur statique. Pour plus d'informations, consultez Constructeurs statiques.

L'exemple suivant présente un constructeur d'instance :

```
class Coords
{
    public int x, y;

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

Pour plus de clarté, cette classe contient des champs publics. L'utilisation de champs publics n'est pas une pratique de programmation recommandée, car elle octroie à n'importe quelle méthode n'importe où dans un programme un accès sans restriction ni vérification aux mécanismes internes d'un objet. Les données membres doivent généralement être privées et doivent être accessibles uniquement via les propriétés et les méthodes de classe.

Ce constructeur d'instance est appelé chaque fois qu'un objet basé sur la classe `Coords` est créé. Un constructeur comme celui-ci, qui n'accepte aucun argument, est un *constructeur sans paramètre*. Toutefois, il est souvent utile de fournir des constructeurs supplémentaires. Par exemple, nous pouvons ajouter un constructeur à la classe `Coords` qui nous permet de spécifier les valeurs initiales pour les données membres :

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Cela permet de créer des objets `Coords` avec des valeurs par défaut ou initiales spécifiques, comme suit :

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

Si une classe n'a pas de constructeur, un constructeur sans paramètre est généré automatiquement, et les valeurs par défaut sont utilisées pour initialiser les champs objet. Par exemple, int est initialisé à 0. Pour plus d'informations sur les valeurs par défaut de type, consultez valeurs par défaut des types C#. Par conséquent,

comme le constructeur sans paramètre de la classe `Coords` initialise toutes les données membres à zéro, il peut être supprimé entièrement sans modification du fonctionnement de la classe. Un exemple complet d'utilisation de plusieurs constructeurs est fourni dans l'exemple 1, plus loin dans cette rubrique, et un exemple d'un constructeur généré automatiquement est fourni dans l'exemple 2.

Les constructeurs d'instances peuvent également servir à appeler les constructeurs d'instances des classes de base. Le constructeur de classe peut appeler le constructeur de la classe de base via l'initialiseur, comme suit :

```
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}
```

Dans cet exemple, la classe `Circle` passe des valeurs représentant le rayon et la hauteur au constructeur fourni par `Shape` duquel `Circle` est dérivé. Un exemple complet d'utilisation de `Shape` et `Circle` s'affiche dans cette rubrique en tant qu'exemple 3.

Exemple 1

L'exemple suivant illustre une classe avec deux constructeurs de classe, l'un sans argument et l'autre avec deux arguments.

```
class Coords
{
    public int x, y;

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
 Coords #1 at (0,0)
 Coords #2 at (5,3)
*/
```

Exemple 2

Dans cet exemple, la classe `Person` n'a aucun constructeur, auquel cas un constructeur sans paramètre est fourni automatiquement, et les champs sont initialisés à leurs valeurs par défaut.

```
public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine("Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:  Name: , Age: 0
```

Notez que la valeur par défaut d' `age` est `0` et que celle de `name` est `null`.

Exemple 3

L'exemple suivant illustre l'utilisation de l'initialiseur de classe de base. La classe `Circle` est dérivée de la classe générale `Shape` et la classe `Cylinder` est dérivée de la classe `Circle`. Le constructeur de chaque classe dérivée utilise son initialiseur de classe de base.

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area of the circle = 19.63
    Area of the cylinder = 86.39
*/
```

Pour obtenir d'autres exemples sur l'appel des constructeurs de classe de base, consultez virtual, override et base.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Constructeurs
- Finaliseurs
- static

Constructeurs privés (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Un constructeur privé est un constructeur d'instance spécial. Il est généralement utilisé dans les classes qui contiennent uniquement des membres statiques. Si une classe a un ou plusieurs constructeurs privés, mais n'a aucun constructeur public, les autres classes (à l'exception des classes imbriquées) ne peuvent pas créer d'instances de cette classe. Par exemple :

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

La déclaration du constructeur vide empêche la génération automatique d'un constructeur sans paramètre. Notez que si vous n'utilisez pas de modificateur d'accès avec le constructeur, le constructeur est toujours privé par défaut. En règle générale, le modificateur private est explicitement utilisé pour spécifier que la classe ne peut pas être instanciée.

Les constructeurs privés servent à empêcher la création d'instances d'une classe quand il n'y a pas de champs ou de méthodes d'instance (par exemple, la classe Math) ou quand une méthode est appelée pour obtenir une instance d'une classe. Si toutes les méthodes dans la classe sont statiques, définissez la classe entière comme statique. Pour plus d'informations , consultez classes statiques et membres de classe statique.

Exemple

L'exemple suivant montre une classe qui utilise un constructeur privé.

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notez que si vous décommentez l'instruction suivante dans l'exemple, une erreur est générée, car le constructeur a un niveau de protection qui le rend inaccessible :

```
// Counter aCounter = new Counter();   // Error
```

Spécification du langage C#

Pour plus d'informations, consultez Constructeurs privés dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Constructeurs
- Finaliseurs
- priv
- public

Constructeurs statiques (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Un constructeur statique est utilisé pour initialiser des données statiques ou pour effectuer une action particulière qui ne doit être effectuée qu'une seule fois. Il est automatiquement appelé avant la création de la première instance ou le référencement d'un membre statique.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Notes

Les constructeurs statiques ont les propriétés suivantes :

- Un constructeur statique n'accepte pas de modificateur d'accès et n'a pas de paramètre.
- Une classe ou struct ne peut pas avoir de constructeur statique.
- Les constructeurs statiques ne peuvent pas être hérités ou surchargés.
- Un constructeur statique ne peut pas être appelé directement et est uniquement destiné à être appelé par le Common Language Runtime (CLR). Il est appelé automatiquement.
- L'utilisateur n'a aucun contrôle sur le moment d'exécution du constructeur statique dans le programme.
- Un constructeur statique est automatiquement appelé pour initialiser la classe avant la création de la première instance ou le référencement d'un membre statique. Un constructeur statique s'exécute avant le constructeur d'instance. Le constructeur statique d'un type est appelé lorsqu'une méthode statique assignée à un événement ou à un délégué est appelée, et non quand elle est assignée. Si des initialiseurs de variable de champ statique sont présents dans la classe du constructeur statique, ils seront exécutés dans l'ordre textuel dans lequel ils apparaissent dans la déclaration de classe, immédiatement avant l'exécution du constructeur statique.
- Si vous ne fournissez pas de constructeur statique pour initialiser les champs statiques, tous les champs statiques sont initialisés à leur valeur par défaut, comme indiqué dans les valeurs par défaut des types C#.
- Si un constructeur statique lève une exception, le runtime ne l'appelle pas une deuxième fois et le type reste non initialisé pour la durée de vie du domaine d'application dans lequel votre programme s'exécute. En règle générale, une exception TypeInitializationException est levée lorsqu'un constructeur statique ne parvient pas à instancier un type ou quand une exception non gérée se produit dans un constructeur statique. Pour les constructeurs statiques implicites qui ne sont pas définis explicitement dans le code source, la résolution des problèmes peut nécessiter l'inspection du code en langage intermédiaire (IL).

- La présence d'un constructeur statique empêche l'ajout de l'attribut de type BeforeFieldInit. Cela limite l'optimisation du runtime.
- Un champ déclaré en tant que `static readonly` ne peut être attribué que dans le cadre de sa déclaration ou dans un constructeur statique. Si un constructeur statique explicite n'est pas obligatoire, initialisez les champs statiques lors de la déclaration plutôt que via un constructeur statique pour une meilleure optimisation du runtime.

> **NOTE**
> S'il n'est pas directement accessible, la présence d'un constructeur statique explicite doit être documentée pour faciliter la résolution des exceptions de l'initialisation.

Usage

- Généralement, on utilise des constructeurs statiques quand la classe utilise un fichier journal et que le constructeur sert à écrire des entrées dans ce fichier.
- Les constructeurs statiques sont également utiles pour la création de classes wrapper pour du code non managé, quand le constructeur peut appeler la méthode `LoadLibrary`.
- Les constructeurs statiques sont également un emplacement pratique pour appliquer des contrôles au moment de l'exécution sur le paramètre de type qui ne peuvent pas être vérifiés au moment de la compilation via des contraintes (contraintes de paramètre de type).

Exemple

Dans cet exemple, la classe `Bus` possède un constructeur statique. Quand la première instance de `Bus` est créée (`bus1`), le constructeur statique est appelé pour initialiser la classe. L'exemple de sortie vérifie que le constructeur statique s'exécute une seule fois, même si deux instances de `Bus` sont créées, et qu'il s'exécute avant le constructeur d'instance.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
```

```
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

spécification du langage C#

Pour plus d'informations, voir la section Constructeurs statiques de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Constructeurs
- Classes statiques et membres de classe statique
- Finaliseurs
- Instructions de conception des constructeurs
- Avertissement de sécurité-CA2121 : les constructeurs statiques doivent être privés

Comment écrire un constructeur de copie (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

C# ne fournit pas de constructeur de copie pour les objets, mais vous pouvez en écrire un vous-même.

Exemple

Dans l'exemple suivant, la `Person` classe définit un constructeur de copie qui prend comme argument une instance de `Person`. Les valeurs des propriétés de l'argument sont assignées aux propriétés de la nouvelle instance de `Person`. Le code contient un autre constructeur de copie qui envoie les propriétés `Name` et `Age` de l'instance que vous voulez copier au constructeur d'instance de la classe.

```
class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

Voir aussi

- ICloneable

- Guide de programmation C#
- Classes et structs
- Constructeurs
- Finaliseurs

Finaliseurs (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Les finaliseurs (également appelés **destructeurs**) servent à effectuer les derniers nettoyages nécessaires lorsqu'une instance de classe est collectée par le récupérateur de mémoire.

Notes

- Les finaliseurs ne peuvent pas être définis dans des structs. Ils sont utilisés uniquement avec les classes.
- Une classe ne peut avoir qu'un seul finaliseur.
- Les finaliseurs ne peuvent pas être hérités ou surchargés.
- Les finaliseurs ne peuvent pas être appelés. Ils sont appelés automatiquement.
- Un finaliseur ne prend pas de modificateur et n'a pas de paramètre.

Par exemple, voici une déclaration de finaliseur pour la classe `Car`.

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Un finaliseur peut aussi être implémenté en tant que définition de corps d'expression, comme l'illustre l'exemple suivant.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

Le finaliseur appelle implicitement Finalize sur la classe de base de l'objet. Ainsi, un appel à un finaliseur est traduit implicitement en ce code :

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Cette conception signifie que la `Finalize` méthode est appelée de manière récursive pour toutes les instances dans la chaîne d'héritage, de la plus dérivée à la moins dérivée.

> **NOTE**
>
> Les finaliseurs vides ne doivent pas être utilisés. Quand une classe contient un finaliseur, une entrée est créée dans la file d'attente `Finalize`. Quand le finaliseur est appelé, le récupérateur de mémoire est appelé pour traiter la file d'attente. Un finaliseur vide entraîne une perte de performances inutile.

Le programmeur n'a aucun contrôle sur le moment où le finaliseur est appelé ; le garbage collector décide quand l'appeler. Le récupérateur de mémoire recherche les objets qui ne sont plus utilisés par l'application. S'il considère qu'un objet peut être finalisé, il appelle le finaliseur (s'il y en a un) et libère la mémoire utilisée pour stocker l'objet.

Dans les applications .NET Framework (mais pas dans les applications .NET Core), les finaliseurs sont également appelés quand le programme se termine.

Il est possible de forcer l'garbage collection en appelant Collect , mais la plupart du temps, cet appel doit être évité, car il peut créer des problèmes de performances.

Utiliser des finaliseurs pour libérer des ressources

En général, C# ne nécessite pas autant de gestion de la mémoire de la part du développeur que de langages qui ne ciblent pas un Runtime avec garbage collection. Cela est dû au fait que le garbage collector .NET gère implicitement l'allocation et la libération de mémoire pour vos objets. Toutefois, quand votre application encapsule des ressources non managées, telles que des fenêtres, des fichiers et des connexions réseau, vous devez utiliser des finaliseurs pour libérer ces ressources. Quand l'objet peut être finalisé, le récupérateur de mémoire exécute la méthode `Finalize` de l'objet.

Libération explicite de ressources

Si votre application utilise une ressource externe coûteuse, nous vous recommandons également de proposer un moyen de libérer explicitement la ressource avant que le récupérateur de mémoire ne libère l'objet. Pour libérer la ressource, implémentez une `Dispose` méthode à partir de l' IDisposable interface qui effectue le nettoyage nécessaire pour l'objet. Cela peut améliorer considérablement les performances de l'application. Même avec ce contrôle explicite sur les ressources, le finaliseur devient un dispositif de protection pour nettoyer les ressources si l'appel à la `Dispose` méthode échoue.

Pour plus d'informations sur le nettoyage des ressources, consultez les articles suivants :

- Nettoyage de ressources non managées
- Implémentation d'une méthode dispose
- using, instruction

Exemple

L'exemple suivant crée trois classes qui forment une chaîne d'héritage. La classe `First` est la classe de base, `Second` est dérivée de `First`, et `Third` est dérivée de `Second`. Toutes trois ont des finaliseurs. Dans `Main`, une instance de la classe la plus dérivée est créée. Quand le programme s'exécute, notez que les finaliseurs des trois classes sont appelés automatiquement, et dans l'ordre, de la plus dérivée à la moins dérivée.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's destructor is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's destructor is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's destructor is called.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}
/* Output (to VS Output Window):
    Third's destructor is called.
    Second's destructor is called.
    First's destructor is called.
*/
```

spécification du langage C#

Pour plus d'informations, voir la section Destructeurs de la spécification du langage C#.

Voir aussi

- IDisposable
- Guide de programmation C#
- Constructeurs
- Garbage collection

Initialiseurs d'objet et de collection (Guide de programmation C#)

02/11/2020 • 12 minutes to read • Edit Online

C# vous permet d'instancier un objet ou une collection et d'effectuer des affectations de membres dans une seule instruction.

Initialiseurs d'objet

Les initialiseurs d'objet vous permettent d'affecter des valeurs aux champs ou propriétés accessibles d'un objet, au moment de sa création, sans devoir appeler un constructeur suivi de ses lignes d'instructions d'assignation. La syntaxe de l'initialiseur de l'objet vous permet de spécifier les arguments d'un constructeur ou de les omettre (et la syntaxe de parenthèses). L'exemple suivant montre comment utiliser l'initialiseur de l'objet de type nommé, `Cat`, et comment appeler le constructeur sans paramètre. Notez l'utilisation de propriétés implémentées automatiquement dans la classe `Cat`. Pour plus d'informations, consultez Propriétés implémentées automatiquement.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

La syntaxe des initialiseurs d'objet vous permet de créer une instance qui assigne l'objet nouvellement créé, avec ses propriétés, à la variable dans l'assignation.

À compter de C# 6, les initialiseurs d'objet peuvent, en plus d'affecter des champs et des propriétés, définir des indexeurs. Prenez l'exemple de cette classe `Matrix` de base :

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Vous pouvez initialiser la matrice identity avec le code suivant :

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Tout indexeur accessible qui contient un setter accessible peut être utilisé comme l'une des expressions d'un initialiseur d'objet, indépendamment du nombre ou des types d'arguments. Les arguments d'index forment le côté gauche de l'affectation, et la valeur le côté droit de l'expression. Par exemple, les éléments suivants sont tous valides si `IndexersExample` a les indexeurs appropriés :

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Pour le code précédent à compiler, le type `IndexersExample` doit avoir les membres suivants :

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

Initialiseurs d'objet avec des types anonymes

Bien que les initialiseurs d'objets puissent être utilisés dans n'importe quel contexte, ils sont particulièrement utiles dans les expressions de requête LINQ. Les expressions de requête utilisent souvent des types anonymes, qui peuvent uniquement être initialisés à l'aide d'un initialiseur d'objet, comme indiqué dans la déclaration suivante.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Les types anonymes permettent `select` à la clause d'une expression de requête LINQ de transformer les objets de la séquence d'origine en objets dont la valeur et la forme peuvent différer de l'original. Cela s'avère utile si vous souhaitez stocker uniquement une partie des informations de chaque objet d'une séquence. Dans l'exemple suivant, supposons qu'un objet de produit (`p`) contient de nombreux champs et méthodes et que vous souhaitez uniquement créer une séquence d'objets qui contiennent le nom de produit et le prix unitaire.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Lorsque cette requête est exécutée, la variable `productInfos` contient une séquence d'objets qui sont accessibles dans une instruction `foreach` comme indiqué dans cet exemple :

```
foreach(var p in productInfos){...}
```

Chaque objet dans le nouveau type anonyme a deux propriétés publiques qui reçoivent les mêmes noms que les propriétés ou champs de l'objet d'origine. Vous pouvez également renommer un champ lorsque vous créez un type anonyme. L'exemple suivant renomme le champ `UnitPrice` en `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Initialiseurs de collection

Les initialiseurs de collection vous permettent de spécifier un ou plusieurs initialiseurs d'élément quand vous initialisez un type de collection qui implémente IEnumerable et qui utilise `Add` comme méthode d'instance ou méthode d'extension avec la signature appropriée. Les initialiseurs d'élément peuvent être une valeur simple, une expression ou un initialiseur d'objet. En utilisant un initialiseur de collection, il n'est pas nécessaire de spécifier plusieurs appels ; le compilateur les ajoute automatiquement.

L'exemple suivant montre deux initialiseurs de collection simples :

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

L'initialiseur de collection suivant utilise des initialiseurs d'objets pour initialiser les objets de la classe `Cat` définis dans un exemple précédent. Notez que les initialiseurs d'objets individuels sont placés entre accolades et séparés par une virgule.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

Vous pouvez spécifier Null comme élément dans un initialiseur de collection si la méthode `Add` de la collection l'autorise.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

Vous pouvez spécifier des éléments indexés si la collection prend en charge l'indexation en lecture/écriture.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

L'exemple précédent génère du code qui appelle Item[TKey] pour définir les valeurs. Avant C# 6, vous pouviez initialiser des dictionnaires et d'autres conteneurs associatifs à l'aide de la syntaxe suivante. Notez que la syntaxe de l'indexeur, avec des parenthèses et une affectation, est remplacée par un objet avec plusieurs valeurs :

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

Cet exemple d'initialiseur appelle Add(TKey, TValue) pour ajouter les trois éléments dans le dictionnaire. Ces deux manières d'initialiser les collections associatives ont un comportement légèrement différent en raison des appels de méthode générés par le compilateur. Les deux variantes fonctionnent avec la classe `Dictionary` . D'autres types peuvent uniquement prendre en charge l'une ou l'autre en fonction de leur API publique.

Exemples

L'exemple suivant combine les concepts d'initialiseurs d'objet et de collection.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

L'exemple suivant montre un objet qui implémente IEnumerable et qui contient une méthode `Add` avec plusieurs paramètres. Il utilise un initialiseur de collection avec plusieurs éléments dans la liste correspondant à la signature de la méthode `Add`.

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

Les méthodes `Add` peuvent utiliser le mot clé `params` pour sélectionner un nombre variable d'arguments, comme indiqué dans l'exemple suivant. Cet exemple montre également l'implémentation personnalisée d'un indexeur pour initialiser une collection à l'aide d'index.

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
```

```
        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
```

```
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
     */
}
```

Voir aussi

- Guide de programmation C#
- LINQ en C#
- Types anonymes

Comment initialiser des objets à l'aide d'un initialiseur d'objet (Guide de programmation C#)

16/12/2020 • 3 minutes to read • Edit Online

Vous pouvez utiliser des initialiseurs d'objets pour initialiser des objets de type de façon déclarative sans appeler explicitement un constructeur pour le type.

Les exemples suivants montrent comment utiliser des initialiseurs d'objets avec des objets nommés. Le compilateur traite les initialiseurs d'objets en accédant tout d'abord au constructeur d'instance sans paramètre, puis en traitant les initialisations de membres. Ainsi, si le constructeur sans paramètre est déclaré comme `private` dans la classe, les initialiseurs d'objets qui nécessitent un accès public échoueront.

Vous devez utiliser un initialiseur d'objet si vous définissez un type anonyme. Pour plus d'informations, consultez Comment retourner des sous-ensembles de propriétés d'éléments dans une requête.

Exemple

L'exemple suivant montre comment initialiser un nouveau type `StudentName` à l'aide d'initialiseurs d'objets. Cet exemple définit des propriétés dans le type `StudentName` :

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
```

```
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Des initialiseurs d'objet peuvent être utilisés pour définir des indexeurs dans un objet. L'exemple suivant définit une classe `BaseballTeam` qui utilise un indexeur pour obtenir et définir des joueurs à des positions différentes. L'initialiseur peut affecter des joueurs en fonction de l'abréviation de la position ou du chiffre utilisé pour désigner la position sur les feuilles de match :

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

Voir aussi

- Guide de programmation C#
- Initialiseurs d'objets et de collections

Guide pratique pour initialiser un dictionnaire avec un initialiseur de collection (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Un Dictionary<TKey,TValue> contient une collection de paires clé/valeur. Sa méthode Add prend deux paramètres, un pour la clé et un pour la valeur. Une façon d'initialiser un Dictionary<TKey,TValue> ou toute collection dont la méthode `Add` prend plusieurs paramètres consiste à mettre chaque jeu de paramètres entre accolades, comme illustré dans l'exemple suivant. Une autre option consiste à utiliser un initialiseur d'index, également illustré dans l'exemple suivant.

Exemple

Dans l'exemple de code suivant, un Dictionary<TKey,TValue> est initialisé avec des instances de type `StudentName`. La première initialisation utilise la méthode `Add` avec deux arguments. Le compilateur génère un appel à `Add` pour chacune des paires de clés `int` et de valeurs `StudentName`. Le deuxième utilise une méthode d'indexeur publique en lecture/écriture de la classe `Dictionary` :

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

Notez les deux paires d'accolades dans chaque élément de la collection dans la première déclaration. Les accolades les plus à l'intérieur encadrent l'initialiseur d'objet pour le `StudentName` , et les accolades les plus extérieures encadrent l'initialiseur pour la paire clé/valeur qui sera ajoutée à `students` Dictionary<TKey,TValue> . Pour finir, l'initialiseur de collection entier pour le dictionnaire est placé entre accolades. Dans la deuxième initialisation, le côté gauche de l'affectation est la clé et le côté droit est la valeur, avec un initialiseur d'objet pour `StudentName`.

Voir aussi

- Guide de programmation C#
- Initialiseurs d'objets et de collections

Types imbriqués (guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Un type défini dans une classe, un struct, un délégué ou une interface est appelé un type imbriqué. Par exemple

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Que le type externe soit une classe, une interface ou un struct, les types imbriqués sont privéspar défaut ; ils sont accessibles uniquement à partir de leur type conteneur. Dans l'exemple précédent, la classe `Nested` est inaccessible aux types externes.

Vous pouvez aussi spécifier un modificateur d'accès pour définir l'accessibilité d'un type imbriqué, comme ceci :

- Les types imbriqués d'une **classe** peuvent être public, protected, internal, protected internal, private ou private protected.

  Cependant, le fait de définir une classe imbriquée `protected`, `protected internal` ou `private protected` à l'intérieur d'une classe sealed a pour effet de générer l'avertissement de compilateur CS0628, « nouveau membre protected déclaré dans une classe sealed ».

- Les types imbriqués d'un **struct** peuvent être public, internal ou private.

L'exemple suivant fait passer le type de la classe `Nested` à public :

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

Le type imbriqué (ou interne) peut accéder au type conteneur (ou externe). Pour accéder au type conteneur, passez-le en tant qu'argument au constructeur du type imbriqué. Par exemple :

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Un type imbriqué a accès à tous les membres qui sont accessibles à son type conteneur. Il peut accéder aux membres privés et protégés du type conteneur, y compris à tous les membres protégés hérités.

Dans la déclaration précédente, le nom complet de classe `Nested` est `Container.Nested`. Il s'agit du nom utilisé pour créer une instance de la classe imbriquée, comme suit :

```
Container.Nested nest = new Container.Nested();
```

Voir aussi

- Guide de programmation C#
- Classes et structs
- Modificateurs d'accès
- Constructeurs

Classes et méthodes partielles (Guide de programmation C#)

02/11/2020 • 12 minutes to read • Edit Online

Il est possible de fractionner la définition d'une classe, d'un struct, d'une interface ou d'une méthode entre plusieurs fichiers sources. Chaque fichier source contient une section de la définition de méthode ou de type, et toutes les parties sont combinées au moment où l'application est compilée.

Classes partielles

Il peut être utile de fractionner une définition de classe dans les situations suivantes :

- Dans des projets volumineux, le fractionnement d'une classe entre plusieurs fichiers distincts permet à plusieurs programmeurs d'y travailler simultanément.
- Quand vous utilisez une source générée automatiquement, vous pouvez ajouter du code à la classe sans avoir à recréer le fichier source. Visual Studio suit cette approche pour créer des formulaires Windows Forms, du code wrapper de service web, etc. Vous pouvez écrire du code qui utilise ces classes sans avoir à modifier le fichier créé par Visual Studio.
- Pour fractionner une définition de classe, utilisez le modificateur de mot clé partial, comme ci-dessous :

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Le mot clé `partial` indique que d'autres parties de la classe, du struct ou de l'interface peuvent être définies dans l'espace de noms. Toutes les parties doivent utiliser le mot clé `partial`. Toutes les parties doivent être disponibles à la compilation pour former le type final. Toutes les parties doivent avoir la même accessibilité : `public`, `private`, etc.

Si une partie est déclarée comme abstract, l'ensemble du type est considéré comme abstract. Si une partie est déclarée comme sealed, l'ensemble du type est considéré comme sealed. Si une partie déclare un type de base, l'ensemble du type hérite de cette classe.

Toutes les parties qui spécifient une classe de base doivent être en accord, mais les parties qui omettent une classe de base héritent tout de même du type de base. Les parties peuvent spécifier des interfaces de base différentes. Le type final implémente alors toutes les interfaces indiquées dans toutes les déclarations partielles. Tous les membres de classe, de struct ou d'interface déclarés dans une définition partielle sont disponibles pour toutes les autres parties. Le type final est la combinaison de toutes les parties au moment de la compilation.

> **NOTE**
> Le modificateur `partial` n'est pas disponible sur les déclarations de délégué ou d'énumération.

L'exemple suivant montre que les types imbriqués peuvent être partiels, même si le type dans lequel ils sont imbriqués n'est pas partiel lui-même.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

Au moment de la compilation, les attributs de définitions de type partiel sont fusionnés. Observez, par exemple, les déclarations suivantes :

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Elles sont équivalentes aux déclarations suivantes :

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

Les éléments suivants sont fusionnés à partir de toutes les définitions de type partiel :

- Commentaires XML
- interfaces
- attributs de paramètre de type générique
- attributs de classe
- membres

Observez, par exemple, les déclarations suivantes :

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Elles sont équivalentes aux déclarations suivantes :

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

Il y a plusieurs règles à respecter quand vous utilisez des définitions de classe partielle :

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être modifiées avec `partial`. Par exemple, les déclarations de classe suivantes génèrent une erreur :

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- Le modificateur `partial` peut uniquement être placé juste avant les mots clés `class`, `struct` ou `interface`.

- Les types partiels imbriqués sont autorisés dans les définitions de type partiel, comme illustré dans l'exemple suivant :

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être définies dans le même assembly et dans le même module (fichier .exe ou .dll). Les définitions partielles ne peuvent pas être fractionnées entre plusieurs modules.

- Le nom de classe et les paramètres de type générique doivent correspondre dans toutes les définitions de type partiel. Les types génériques peuvent être partiels. Chaque déclaration partielle doit utiliser les mêmes noms de paramètres, dans le même ordre.

- Les mots clés suivants sont facultatifs dans une définition de type partiel. S'ils sont utilisés dans une définition de type partiel, ils ne doivent pas être en conflit avec les mots clés spécifiés dans une autre définition partielle pour le même type :

  - public
  - priv
  - protected
  - intérieurs
  - abstraction
  - sealed
  - classe de base
  - modificateur new (parties imbriquées)
  - contraintes génériques

Pour plus d'informations, consultez Contraintes sur les paramètres de type.

Exemple 1

Description

Dans l'exemple suivant, les champs et le constructeur de la classe `Coords` sont déclarés dans une définition de classe partielle, et le membre `PrintCoords` est déclaré dans une autre définition de classe partielle.

Code

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

Exemple 2

Description

L'exemple suivant montre que vous pouvez également développer des interfaces et des structs partiels.

Code

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Méthodes partielles

Une classe partielle ou un struct partiel peut contenir une méthode partielle. Une partie de la classe contient la signature de la méthode. Une implémentation facultative peut être définie dans la même partie ou dans une autre partie. Si l'implémentation n'est pas fournie, la méthode et tous les appels à cette méthode sont supprimés au moment de la compilation.

Les méthodes partielles permettent à l'implémenteur d'une partie d'une classe de définir une méthode, comme pour un événement. L'implémenteur de l'autre partie de la classe peut décider s'il faut implémenter la méthode ou non. Si la méthode n'est pas implémentée, le compilateur supprime la signature de méthode et tous les appels à la méthode. Les appels à la méthode, y compris tous les résultats retournés par l'évaluation des arguments dans les appels, n'ont aucun effet au moment de l'exécution. C'est pourquoi le code dans la classe partielle peut utiliser librement une méthode partielle, même si l'implémentation n'est pas fournie. Aucune erreur de compilation ou d'exécution n'est générée si la méthode est appelée, mais qu'elle n'est pas finalement implémentée.

Les méthodes partielles sont particulièrement utiles pour personnaliser le code généré. Elles permettent de réserver un nom et une signature de méthode. Ainsi, le code généré peut appeler la méthode, mais le développeur peut décider d'implémenter ou non la méthode. En grande partie comme les classes partielles, les méthodes partielles permettent d'utiliser conjointement du code créé par un générateur de code et du code créé manuellement par un développeur sans impact sur le temps d'exécution.

Une déclaration de méthode partielle se compose de deux parties : la définition et l'implémentation. Celles-ci peuvent être situées dans des parties distinctes ou dans la même partie d'une classe partielle. En l'absence de déclaration d'implémentation, le compilateur optimise à la fois la déclaration de définition et tous les appels à la méthode.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- Une déclaration de méthode partielle doit commencer par le mot clé contextuel partial, et la méthode doit retourner void.

- Les méthodes partielles peuvent avoir des paramètres in ou ref, mais pas out.
- Les méthodes partielles sont implicitement private et, par conséquent, elles ne peuvent pas être virtual.
- Les méthodes partielles ne peuvent pas être extern, car la présence du corps détermine si elles sont utilisées pour la définition ou pour l'implémentation.
- Les méthodes partielles peuvent avoir des modificateurs static et unsafe.
- Les méthodes partielles peuvent être génériques. Les contraintes sont placées sur la déclaration de méthode partielle de définition et peuvent être répétées facultativement sur la déclaration d'implémentation. Les noms de paramètre et de paramètre de type ne doivent pas obligatoirement être identiques dans la déclaration d'implémentation et la déclaration de définition.
- Vous pouvez créer un délégué pour une méthode partielle qui a été définie et implémentée, mais pas pour une méthode partielle qui a uniquement été définie.

Spécification du langage C#

Pour plus d'informations, consultez Types partiels dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Catégories
- Types de structure
- Interfaces
- partial, Type

Types anonymes (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Les types anonymes permettent d'encapsuler un ensemble de propriétés en lecture seule dans un unique objet sans avoir à définir explicitement un type. Le nom du type est généré par le compilateur et n'est pas disponible au niveau du code source. Le type de chaque propriété est déduit par le compilateur.

Vous créez des types anonymes en utilisant l'opérateur new avec un initialiseur d'objet. Pour plus d'informations sur les initialiseurs d'objets, consultez Initialiseurs d'objets et de collections.

L'exemple suivant montre un type anonyme qui est initialisé avec deux propriétés nommées `Amount` et `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Les types anonymes sont en règle générale utilisés dans la clause select d'une expression de requête pour retourner un sous-ensemble de propriétés de chaque objet dans la séquence source. Pour plus d'informations sur les requêtes, consultez LINQ en C#.

Les types anonymes contiennent une ou plusieurs propriétés en lecture seule publiques. Aucun autre type de membres de classe, tels que des méthodes ou des événements, n'est valide. L'expression qui est utilisée pour initialiser une propriété ne peut pas être `null`, une fonction anonyme ou un type pointeur.

Le scénario le plus courant consiste à initialiser un type anonyme avec les propriétés d'un autre type. Dans l'exemple suivant, une classe existe qui porte le nom `Product`. La classe `Product` comprend des propriétés `Color` et `Price`, ainsi que d'autres propriétés qui ne vous intéressent pas. La variable `products` est une collection d'objets `Product`. La déclaration de type anonyme commence par le mot clé `new`. La déclaration initialise un nouveau type qui utilise uniquement deux propriétés de `Product`. Une quantité réduite de données est ainsi retournée dans la requête.

Si vous n'indiquez pas les noms de membre dans le type anonyme, le compilateur attribue aux membres de type anonyme le même nom que celui de la propriété utilisée pour les initialiser. Vous devez fournir un nom pour une propriété qui est initialisée avec une expression, comme indiqué dans l'exemple précédent. Dans l'exemple suivant, les noms des propriétés du type anonyme sont `Color` et `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Quand vous utilisez un type anonyme pour initialiser une variable, vous déclarez la variable en tant que variable locale implicitement typée en utilisant var. Le nom de type ne peut pas être spécifié dans la déclaration de variable, car seul le compilateur a accès au nom sous-jacent du type anonyme. Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.

Vous pouvez créer un tableau d'éléments typés anonymement en associant une variable locale implicitement typée et un tableau typé implicitement, comme indiqué dans l'exemple suivant.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Notes

Les types anonymes sont des types class qui dérivent directement d'object et qui ne peuvent pas être castés en type à l'exception du type object. Le compilateur fournit un nom pour chaque type anonyme, bien que votre application ne puisse pas y accéder. Du point de vue du CLR, un type anonyme n'est pas différent des autres types de référence.

Si plusieurs initialiseurs d'objet dans un assembly spécifient une séquence de propriétés dans le même ordre et qui sont du même type et portent le même nom, le compilateur traite les objets comme des instances du même type. Elles partagent les mêmes informations de type générées par le compilateur.

Vous ne pouvez pas déclarer un champ, une propriété, un événement ou le type de retour d'une méthode comme ayant un type anonyme. De même, vous ne pouvez pas déclarer un paramètre formel d'une méthode, d'une propriété, d'un constructeur ou d'un indexeur comme ayant un type anonyme. Pour passer un type anonyme, ou une collection qui contient des types anonymes, en tant qu'argument d'une méthode, vous pouvez déclarer le paramètre comme objet de type. Cependant, cela va à l'encontre de l'objectif du typage fort. Si vous devez stocker les résultats de requête ou les passer en dehors des limites de la méthode, utilisez un struct ordinaire ou une classe au lieu d'un type anonyme.

Dans la mesure où les méthodes Equals et GetHashCode dans les types anonymes sont définies selon les termes des méthodes `Equals` et `GetHashCode` des propriétés, deux instances du même type anonyme sont égales si toutes leurs propriétés sont égales.

Voir aussi

- Guide de programmation C#
- Initialiseurs d'objets et de collections
- Mise en route de LINQ en C#
- LINQ en C#

Comment retourner des sous-ensembles de propriétés d'éléments dans une requête (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Utilisez un type anonyme dans une expression de requête lorsque les deux conditions suivantes s'appliquent :

- Vous souhaitez retourner uniquement certaines propriétés de chaque élément source.
- Vous n'avez pas besoin de stocker les résultats de requête en dehors de la portée de la méthode dans laquelle la requête est exécutée.

Si vous souhaitez uniquement retourner une propriété ou un champ de chaque élément source, vous pouvez utiliser l'opérateur point dans la clause `select`. Par exemple, pour retourner uniquement l'`ID` de chaque `student`, écrivez la clause `select` de la façon suivante :

```
select student.ID;
```

Exemple

L'exemple suivant montre comment utiliser un type anonyme pour retourner uniquement un sous-ensemble des propriétés de chaque élément source qui répond à la condition spécifiée.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Notez que le type anonyme utilise les noms de l'élément source pour ses propriétés si aucun nom n'est spécifié. Pour attribuer de nouveaux noms aux propriétés du type anonyme, écrivez l'instruction `select` de la façon suivante :

```
select new { First = student.FirstName, Last = student.LastName };
```

Si vous effectuez cette opération dans l'exemple précédent, l'instruction `Console.WriteLine` doit également changer :

```
Console.WriteLine(student.First + " " + student.Last);
```

Compilation du code

Pour exécuter ce code, copiez et collez la classe dans une application console C# avec une directive `using` pour System.Linq.

Voir aussi

- Guide de programmation C#
- Types anonymes
- LINQ en C#

Interfaces (Guide de programmation C#)

02/11/2020 • 9 minutes to read • Edit Online

Une interface contient des définitions pour un groupe de fonctionnalités connexes qu'une classe non abstraite ou un struct doit implémenter. Une interface peut définir des `static` méthodes, qui doivent avoir une implémentation. À compter de C# 8,0, une interface peut définir une implémentation par défaut pour les membres. Une interface ne peut pas déclarer des données d'instance, telles que des champs, des propriétés implémentées automatiquement ou des événements de type propriété.

En utilisant des interfaces, vous pouvez, par exemple, inclure le comportement de plusieurs sources dans une classe. Cette fonctionnalité est importante en C#, car le langage ne prend pas en charge l'héritage multiple de classes. De plus, vous devez utiliser une interface si vous voulez simuler l'héritage pour des structs, car ils ne peuvent en fait pas hériter d'un autre struct ou d'une autre classe.

Vous définissez une interface à l'aide du mot clé interface , comme le montre l'exemple suivant.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Le nom d'une interface doit être un nom d' identificateurC# valide. Par convention, les noms d'interface commencent par un `I` majuscule.

Toute classe ou tout struct qui implémentent l'interface IEquatable<T> doivent contenir une définition pour une méthode Equals qui correspond à la signature spécifiée par l'interface. Ainsi, vous pouvez compter sur une classe qui implémente `IEquatable<T>` pour contenir une méthode `Equals` avec laquelle une instance de la classe peut déterminer si elle est égale à une autre instance de la même classe.

La définition de `IEquatable<T>` ne fournit pas d'implémentation pour `Equals` . Une classe ou un struct peut implémenter plusieurs interfaces, mais une classe peut uniquement hériter d'une classe unique.

Pour plus d'informations sur les classes abstraites, consultez Classes abstract et sealed et membres de classe.

Les interfaces peuvent contenir des méthodes d'instance, des propriétés, des événements, des indexeurs ou toute combinaison de ces quatre types de membres. Les interfaces peuvent contenir des constructeurs statiques, des champs, des constantes ou des opérateurs. Pour obtenir des liens vers des exemples, consultez les Sections connexes. Une interface ne peut pas contenir de champs d'instance, de constructeurs d'instance ou de finaliseurs. Les membres d'interface sont publics par défaut.

Pour implémenter un membre d'interface, le membre correspondant de la classe d'implémentation doit être public, non statique et porter le même nom et la même signature que le membre d'interface.

Quand une classe ou un struct implémente une interface, la classe ou le struct doivent fournir une implémentation pour tous les membres que l'interface déclare, mais ne fournit pas d'implémentation par défaut pour. Toutefois, si une classe de base implémente une interface, toute classe dérivée de la classe de base hérite de cette implémentation.

L'exemple suivant illustre une implémentation de l'interface IEquatable<T>. La classe d'implémentation, `Car`, doit fournir une implémentation de la méthode Equals.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Les propriétés et indexeurs d'une classe peuvent définir des accesseurs supplémentaires pour une propriété ou un indexeur qui est défini dans une interface. Par exemple, une interface peut déclarer une propriété qui a un accesseur get. La classe qui implémente l'interface peut déclarer la même propriété avec à la fois un accesseur `get` et un accesseur set. Toutefois, si la propriété ou l'indexeur utilisent une implémentation explicite, les accesseurs doivent correspondre. Pour plus d'informations sur l'implémentation explicite, consultez les pages Implémentation d'interface explicite et Propriétés d'interface.

Les interfaces peuvent hériter d'une ou de plusieurs interfaces. L'interface dérivée hérite des membres de ses interfaces de base. Une classe qui implémente une interface dérivée doit implémenter tous les membres de l'interface dérivée, y compris tous les membres des interfaces de base de l'interface dérivée. Cette classe peut être implicitement convertie en une interface dérivée ou l'une de ses interfaces de base. Une classe peut inclure une interface plusieurs fois par le biais de classes de base qu'elle hérite ou d'interfaces dont d'autres interfaces héritent. Toutefois, la classe peut fournir une implémentation d'une interface une seule fois et uniquement si la classe déclare l'interface dans le cadre de la définition de la classe (`class ClassName : InterfaceName`). Si l'interface est héritée car vous avez hérité d'une classe de base qui implémente l'interface, la classe de base fournit l'implémentation des membres de l'interface. Toutefois, la classe dérivée peut réimplémenter des membres d'interface virtuels au lieu d'utiliser l'implémentation héritée. Quand les interfaces déclarent une implémentation par défaut d'une méthode, toute classe qui implémente cette interface hérite de cette implémentation. Les implémentations définies dans les interfaces sont virtuelles et la classe d'implémentation peut remplacer cette implémentation.

Une classe de base peut également implémenter des membres d'interface à l'aide de membres virtuels. Dans ce cas, une classe dérivée peut modifier le comportement de l'interface en substituant les membres virtuels. Pour plus d'informations sur les membres virtuels, consultez la page Polymorphisme.

Résumé des interfaces

Une interface possède les propriétés suivantes :

- Une interface est généralement similaire à une classe de base abstraite avec des membres abstraits uniquement. Toute classe ou tout struct qui implémentent l'interface doivent implémenter tous ses membres. Éventuellement, une interface peut définir des implémentations par défaut pour tout ou partie de ses membres. Pour plus d'informations, consultez méthodes d'interface par défaut.
- Une interface ne peut pas être instanciée directement. Ses membres sont implémentées par une classe ou un struct qui implémentent l'interface.
- Une classe ou un struct peuvent implémenter plusieurs interfaces. Une classe peut hériter d'une classe de base et également implémenter une ou plusieurs interfaces.

Sections connexes

- Propriétés de l'interface

- Indexeurs dans les interfaces
- Comment implémenter des événements d'interface
- Classes et structs
- Héritage
- Interfaces
- Méthodes
- Polymorphisme
- Classes abstract et sealed et membres de classe
- Propriétés
- Événements
- Indexeurs

Voir aussi

- Guide de programmation C#
- Héritage
- Noms d'identificateurs

Implémentation d'interface explicite (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Si une classe implémente deux interfaces qui contiennent un membre avec la même signature, l'implémentation de ce membre sur la classe a pour conséquence que les deux interfaces utilisent ce membre comme leur implémentation. Dans l'exemple suivant, tous les appels à `Paint` appellent la même méthode. Ce premier exemple définit les types :

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

L'exemple suivant appelle les méthodes :

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();
// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Lorsque deux membres d'interface n'exécutent pas la même fonction, il en résulte une implémentation incorrecte de l'une ou des deux interfaces. Il est possible d'implémenter un membre d'interface explicitement, en créant un membre de classe qui est appelé uniquement par le biais de l'interface, et est spécifique à cette interface. Nommez le membre de classe avec le nom de l'interface et un point. Par exemple :

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Le membre de classe `IControl.Paint` est disponible uniquement via l'interface `IControl` et `ISurface.Paint` est disponible uniquement via `ISurface`. Les deux implémentations de méthode sont distinctes et ne sont pas directement disponibles sur la classe. Par exemple :

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

L'implémentation explicite est également utilisée pour résoudre les cas où deux interfaces déclarent chacune des membres différents du même nom, tels qu'une propriété et une méthode. Pour implémenter les deux interfaces, une classe doit utiliser l'implémentation explicite pour la propriété `P`, la méthode `P`, ou les deux, pour éviter une erreur du compilateur. Par exemple :

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

À compter de C# 8,0, vous pouvez définir une implémentation pour les membres déclarés dans une interface. Si une classe hérite d'une implémentation de méthode d'une interface, cette méthode est uniquement accessible par le biais d'une référence du type interface. Le membre hérité n'apparaît pas dans le cadre de l'interface publique. L'exemple suivant définit une implémentation par défaut pour une méthode d'interface :

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

L'exemple suivant appelle l'implémentation par défaut :

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Toute classe qui implémente l' `IControl` interface peut substituer la méthode par défaut `Paint` , en tant que méthode publique, ou en tant qu'implémentation d'interface explicite.

Voir aussi

- Guide de programmation C#
- Classes et structs
- Interfaces
- Héritage

Comment implémenter de manière explicite des membres d'interface (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple déclare une interface, `IDimensions` , et une classe, `Box` , qui implémente explicitement les membres d'interface `GetLength` et `GetWidth` . Les membres sont accessibles via l'instance d'interface `dimensions` .

Exemple

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Programmation fiable

- Notez que les lignes suivantes de la méthode `Main` sont commentées car elles produiraient des erreurs de compilation. Un membre d'interface qui est implémenté de façon explicite n'est pas accessible à partir d'une instance de classe :

  ```
  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- Notez également que les lignes suivantes de la méthode `Main` impriment correctement les dimensions de la zone, car les méthodes sont appelées à partir d'une instance de l'interface :

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

Voir aussi

- Guide de programmation C#
- Classes et structs
- Interfaces
- Comment implémenter explicitement des membres de deux interfaces

Comment implémenter explicitement des membres de deux interfaces (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

L'implémentation explicite d'une interface permet également au programmeur d'implémenter deux interfaces qui ont les mêmes noms de membres et donnent à chaque membre d'interface une implémentation distincte. L'exemple suivant affiche les dimensions d'une zone dans les unités de mesure à la fois métriques et britanniques. La classe Box implémente deux interfaces, IEnglishDimensions et IMetricDimensions, qui représentent les différents systèmes de mesure. Les deux interfaces ont des noms de membres identiques, Length et Width.

Exemple

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Programmation fiable

Si vous souhaitez utiliser par défaut les unités de mesure britanniques, implémentez les méthodes Length et Width normalement, puis implémentez explicitement les méthodes Length et Width à partir de l'interface IMetricDimensions :

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

Dans ce cas, vous pouvez accéder aux unités de mesure britanniques à partir de l'instance de classe et accéder aux unités métriques à partir de l'instance d'interface :

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

Voir aussi

- Guide de programmation C#
- Classes et structs
- Interfaces
- Comment implémenter explicitement des membres d'interface

Délégués (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

Un délégué est un type qui représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Lorsque vous instanciez un délégué, vous pouvez associer son instance à toute méthode ayant une signature et un type de retour compatibles. Vous pouvez appeler la méthode par le biais l'instance de délégué.

Les délégués sont utilisés pour passer des méthodes comme arguments à d'autres méthodes. Les gestionnaires d'événements sont tout simplement des méthodes appelées par le biais de délégués. Vous créez une méthode personnalisée, et une classe telle qu'un contrôle Windows peut appeler votre méthode lorsqu'un certain événement se produit. L'exemple suivant illustre une déclaration de délégué :

```
public delegate int PerformCalculation(int x, int y);
```

Toute méthode de n'importe quelle classe ou structure accessible qui correspond au type de délégué, peut être assignée au délégué. La méthode peut être une méthode d'instance ou statique. Cela permet de modifier par programme les appels de méthode, mais également d'insérer du nouveau code dans les classes existantes.

> **NOTE**
> Dans le contexte de surcharge de méthodes, la signature d'une méthode n'inclut pas la valeur de retour. Mais dans le contexte des délégués, la signature inclut la valeur de retour. En d'autres termes, une méthode doit avoir le même type de retour que le délégué.

Cette capacité à faire référence à une méthode en tant que paramètre fait des délégués les instruments idéaux pour définir des méthodes de rappel. Par exemple, une référence à une méthode qui compare deux objets peut être passée comme argument à un algorithme de tri. Étant donné que le code de comparaison est dans une procédure distincte, l'algorithme de tri peut être écrit de façon plus générale.

Vue d'ensemble des délégués

Les délégués ont les propriétés suivantes :

- Les délégués sont comparables aux pointeurs de fonction C++, sauf que les délégués sont totalement orientés objet, et contrairement aux pointeurs C++ vers les fonctions membres, les délégués encapsulent une instance d'objet et une méthode.
- Les délégués permettent aux méthodes d'être transmises comme des paramètres.
- Les délégués peuvent être utilisés pour définir des méthodes de rappel.
- Les délégués peuvent être chaînés ; par exemple, plusieurs méthodes peuvent être appelées sur un seul événement.
- Les méthodes ne doivent pas correspondre exactement au type du délégué. Pour plus d'informations, consultez Utilisation de la variance dans les délégués.
- C# version 2.0 a introduit le concept de méthodes anonymes, qui permet de passer des blocs de code en tant que paramètres à la place d'une méthode définie séparément. C# 3.0 a introduit les expressions lambda comme un moyen plus concis d'écrire des blocs de code inline. Les méthodes anonymes et les

expressions lambda (dans certains contextes) sont toutes deux compilées en types de délégué. Ces fonctionnalités sont désormais conjointement désignées par l'expression « fonctions anonymes ». Pour plus d'informations sur les expressions lambda, voir Expressions lambda.

Dans cette section

- Utilisation de délégués
- Quand utiliser des délégués à la place d'interfaces (Guide de programmation C#)
- Délégués avec méthodes nommées et Méthodes anonymes
- Utilisation de la variance dans les délégués
- Comment combiner des délégués (délégués multicast)
- Comment déclarer, instancier et utiliser un délégué

Spécification du langage C#

Pour plus d'informations, consultez Délégués dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Chapitres proposés

Delegates, Events, and Lambda Expressions (Délégués, événements et expressions lambda) dans C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events dans Learning C# 3.0: Master the fundamentals of C# 3.0

Voir aussi

- Delegate
- Guide de programmation C#
- Événements

Utilisation de délégués (guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Un délégué est un type qui encapsule sans risque une méthode ; il est similaire à un pointeur de fonction en C et C++. Toutefois, à la différence des pointeurs de fonction C, les délégués sont orientés objet, de type sécurisé et sûrs. Le type d'un délégué est défini par le nom du délégué. L'exemple suivant déclare un délégué nommé `Del` qui peut encapsuler une méthode acceptant une chaîne (string) comme argument et qui retourne void :

```
public delegate void Del(string message);
```

Un objet de délégué est normalement construit en fournissant le nom de la méthode que le délégué encapsule, ou avec une fonction anonyme. Une fois qu'un délégué est instancié, un appel de méthode fait au délégué est transmis par le délégué à cette méthode. Les paramètres passés au délégué par l'appelant sont passés à la méthode et la valeur de retour de la méthode, le cas échéant, est retournée à l'appelant par le délégué. Cette opération est connue sous le nom d'appel du délégué. Un délégué instancié peut être appelé comme s'il s'agissait de la méthode encapsulée elle-même. Par exemple :

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Les types délégués sont dérivés de la Delegate classe dans .net. Les types délégués sont scellés (sealed), c'est-à-dire qu'ils ne peuvent pas être faire l'objet d'une dérivation, et il n'est pas possible de dériver des classes personnalisées à partir de Delegate. Étant donné que le délégué instancié est un objet, il peut être passé comme paramètre ou assigné à une propriété. Cela permet à une méthode d'accepter un délégué comme paramètre et d'appeler le délégué ultérieurement. Cette opération est connue sous le nom de rappel asynchrone et constitue une méthode courante pour notifier un appelant quand un long processus s'est achevé. Lorsqu'un délégué est ainsi utilisé, le code qui utilise le délégué n'a pas besoin de connaître l'implémentation de la méthode utilisée. La fonctionnalité est semblable à l'encapsulation que fournissent les interfaces.

Une autre utilisation courante des rappels consiste à définir une méthode de comparaison personnalisée et à passer ce délégué à une méthode de tri. Le code de l'appelant peut ainsi faire partie de l'algorithme de tri. L'exemple de méthode suivant utilise le type `Del` comme paramètre :

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Vous pouvez ensuite passer le délégué créé ci-dessus à cette méthode :

```
MethodWithCallback(1, 2, handler);
```

et vous recevez le résultat suivant sur la console :

```
The number is: 3
```

En utilisant le délégué comme abstraction, `MethodWithCallback` n'a pas besoin d'appeler la console directement. La méthode n'a pas à être conçue en l'associant obligatoirement à une console. En effet, `MethodWithCallback` se contente de préparer une chaîne et de la passer à une autre méthode. Cela est particulièrement efficace, car une méthode déléguée peut utiliser n'importe quel nombre de paramètres.

Quand un délégué est construit pour encapsuler une méthode d'instance, le délégué fait référence à l'instance et à la méthode. Un délégué n'ayant aucune connaissance du type d'instance hormis la méthode qu'il encapsule, un délégué peut faire référence à n'importe quel type d'objet tant qu'il existe une méthode sur l'objet correspondant à la signature du délégué. Quand un délégué est construit pour encapsuler une méthode statique, il fait uniquement référence à la méthode. Prenons les déclarations suivantes :

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Avec la méthode statique `DelegateMethod` montrée précédemment, nous avons désormais trois méthodes pouvant être encapsulées par une instance `Del`.

Un délégué peut appeler plusieurs méthodes quand il est appelé. Cette opération se nomme multidiffusion. L'ajout d'une méthode supplémentaire à la liste des méthodes du délégué – la liste d'invocation – nécessite simplement l'ajout de deux délégués à l'aide des opérateurs d'addition ou d'assignation d'addition (« + » ou « += »). Par exemple :

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

À ce stade, `allMethodsDelegate` contient trois méthodes dans sa liste d'invocation : `Method1`, `Method2` et `DelegateMethod`. Les trois délégués initiaux, `d1`, `d2` et `d3`, restent inchangés. Lors de l'appel de `allMethodsDelegate`, les trois méthodes sont appelées dans l'ordre. Si le délégué utilise des paramètres de référence, la référence est passée séquentiellement à chacune des trois méthodes et toute modification apportée par une méthode est visible dans la méthode suivante. Quand l'une des méthodes lève une exception qui n'est pas interceptée dans la méthode, cette exception est passée à l'appelant du délégué et aucune des méthodes suivantes dans la liste d'invocation n'est appelée. Si le délégué a une valeur de retour et/ou des paramètres out, il retourne la valeur de retour et les paramètres de la dernière méthode appelée. Pour supprimer une méthode de la liste d'appel, utilisez les opérateurs d'assignation de soustraction ou de soustraction (`-` ou `-=`). Par exemple :

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Étant donné que les types délégués sont dérivés de `System.Delegate`, les méthodes et les propriétés définies par cette classe peuvent être appelées sur le délégué. Par exemple, pour rechercher le nombre de méthodes dans la liste d'invocation d'un délégué, vous pouvez écrire :

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Les délégués ayant plusieurs méthodes dans leur liste d'invocation dérivent de MulticastDelegate, qui est une sous-classe de `System.Delegate`. Le code ci-dessus fonctionne dans les deux cas, car les deux classes prennent en charge `GetInvocationList`.

Les délégués multicast sont largement utilisés dans la gestion des événements. Les objets sources d'événements envoient des notifications d'événement aux objets destinataires qui se sont inscrits pour recevoir cet événement. Pour s'inscrire à un événement, le destinataire crée une méthode conçue pour gérer l'événement, puis crée un délégué pour cette méthode et passe le délégué à la source d'événements. La source appelle le délégué quand l'événement se produit. Le délégué appelle ensuite la méthode de gestion d'événement sur le destinataire, en fournissant les données d'événement. Le type délégué d'un événement donné est défini par la source de l'événement. Pour plus d'informations, consultez Événements.

La comparaison de délégués de deux types différents assignés au moment de la compilation provoque une erreur de compilation. Si les instances de délégué sont statiquement du type `System.Delegate`, la comparaison est alors autorisée, mais retourne la valeur false à l'exécution. Par exemple :

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

Voir aussi

- Guide de programmation C#
- Délégués
- Utilisation de la variance dans les délégués
- Variance dans les délégués
- Utilisation de la variance pour les délégués génériques Func et Action
- Événements

Délégués avec méthodes nommées et méthodes anonymes (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Un délégué peut être associé à une méthode nommée. Quand vous instanciez un délégué à l'aide d'une méthode nommée, la méthode est passée en tant que paramètre, par exemple :

```
// Declare a delegate.
delegate void Del(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
Del d = obj.DoWork;
```

C'est ce qui s'appelle utiliser une méthode nommée. Les délégués construits avec une méthode nommée peuvent encapsuler une méthode static ou une méthode d'instance. Les méthodes nommées représentent la seule façon d'instancier un délégué dans les versions précédentes de C#. Toutefois, dans une situation où la création d'une méthode constitue une charge de travail non souhaitée, C# vous permet d'instancier un délégué et de spécifier immédiatement un bloc de code que le délégué traite quand il est appelé. Le bloc peut contenir une expression lambda ou une méthode anonyme. Pour plus d'informations, consultez Fonctions anonymes.

Notes

La méthode que vous passez en tant que paramètre de délégué doit avoir la même signature que la déclaration du délégué.

Une instance de délégué encapsule soit une méthode statique soit une méthode d'instance.

Même si le délégué peut utiliser un paramètre out, nous ne recommandons pas son utilisation avec les délégués d'événement multicast, car vous ne pouvez pas savoir quel délégué sera appelé.

Exemple 1

L'exemple suivant est un exemple simple de déclaration et d'utilisation d'un délégué. Notez que le délégué, `Del`, et la méthode associée, `MultiplyNumbers`, ont la même signature

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

Exemple 2

Dans l'exemple suivant, un délégué est associé à une méthode statique et à une méthode d'instance, et retourne des informations spécifiques à partir de chacune.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

Voir aussi

- Guide de programmation C#
- Délégués
- Comment combiner des délégués (délégués multicast)
- Événements

Comment combiner des délégués (délégués multicast) (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple explique comment créer des délégués multicast. Une propriété utile des objets délégués est que plusieurs objets peuvent être assignés à une instance de délégué à l'aide de l'opérateur `+`. Le délégué multicast contient une liste des délégués assignés. Quand le délégué multicast est appelé, il appelle les délégués dans la liste, dans l'ordre. Seuls des délégués de même type peuvent être combinés.

Vous pouvez utiliser l'opérateur `-` pour supprimer un délégué de composant d'un délégué multicast.

Exemple

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Voir aussi

- MulticastDelegate
- Guide de programmation C#
- Événements

Comment déclarer, instancier et utiliser un délégué (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Dans C# 1.0 et versions ultérieures, vous pouvez déclarer des délégués comme illustré dans l'exemple suivant.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 offre un moyen plus simple d'écrire la déclaration précédente, comme illustré dans l'exemple suivant.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

Dans C# 2.0 et versions ultérieures, vous pouvez aussi utiliser une méthode anonyme pour déclarer et initialiser un délégué, comme illustré dans l'exemple suivant.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

Dans C# 3.0 et versions ultérieures, vous pouvez aussi déclarer et instancier des délégués à l'aide d'une expression lambda, comme illustré dans l'exemple suivant.

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for: {name}"); };
```

Pour plus d'informations, consultez expressions lambda.

L'exemple suivant illustre la déclaration, l'instanciation et l'utilisation d'un délégué. La classe `BookDB` encapsule une base de données de librairie qui tient à jour un inventaire des livres. Elle expose une méthode, `ProcessPaperbackBooks`, qui recherche tous les livres de poche dans la base de données et appelle un délégué pour chacun d'entre eux. Le type `delegate` qui est utilisé se nomme `ProcessBookDelegate`. La classe `Test` utilise cette classe pour imprimer les titres et le prix moyen des livres de poche.

L'utilisation des délégués favorise une bonne séparation des fonctionnalités entre la base de données de librairie et le code client. Le code client n'a aucune connaissance de la façon dont les livres sont stockés, ni de la façon dont le code librairie trouve les livres de poche. Le code librairie n'a aucune connaissance du traitement effectué sur les livres de poche une fois qu'il les a trouvés.

Exemple

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookDelegate(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
```

```
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Programmation fiable

- Déclaration d’un délégué.

L'instruction suivante déclare un nouveau type délégué.

```
public delegate void ProcessBookDelegate(Book book);
```

Chaque type délégué décrit le nombre et les types des arguments, ainsi que le type de la valeur de retour des méthodes qu'il peut encapsuler. Chaque fois qu'un nouvel ensemble de types d'arguments ou de types de valeur de retour est nécessaire, un nouveau type délégué doit être déclaré.

- Instanciation d'un délégué.

  Une fois que vous avez déclaré un type délégué, vous devez créer un objet délégué et l'associer à une méthode particulière. Pour cela, dans l'exemple précédent vous passez la méthode `PrintTitle` à la méthode `ProcessPaperbackBooks`, comme illustré dans l'exemple suivant :

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  Cela crée un nouvel objet délégué associé à la méthode statique `Test.PrintTitle`. De même, la méthode non statique `AddBookToTotal` sur l'objet `totaller` est passée comme dans l'exemple suivant :

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  Dans les deux cas, un nouvel objet délégué est passé à la méthode `ProcessPaperbackBooks`.

  Une fois un délégué créé, la méthode à laquelle il est associé ne change jamais. Les objets délégué sont immuables.

- Appel d'un délégué.

  Une fois créé, un objet délégué est généralement passé à un autre code qui appellera le délégué. Pour appeler un objet délégué, vous devez utiliser le nom de l'objet délégué, suivi des arguments entre parenthèses à passer au délégué. Voici un exemple d'appel de délégué :

  ```
  processBook(b);
  ```

  Vous pouvez appeler un délégué de manière synchrone, comme dans cet exemple, ou de manière asynchrone à l'aide des méthodes `BeginInvoke` et `EndInvoke`.

Voir aussi

- Guide de programmation C#
- Événements
- Délégués

Tableaux (guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` en tant que type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
type[] arrayName;
```

Exemple

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Présentation du tableau

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions et la longueur de chaque dimension sont établis lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Les valeurs par défaut des éléments de tableau numériques sont définies sur zéro et les éléments de référence sont définis sur Null.
- Un tableau en escalier est un tableau de tableaux, et par conséquent ses éléments sont des types référence et sont initialisés sur `null`.

- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Étant donné que ce type implémente IEnumerable et IEnumerable<T>, vous pouvez utiliser l'itération foreach sur tous les tableaux en C#.

Sections connexes

- Tableaux comme objets
- Utiliser foreach avec des tableaux
- Passage de tableaux en tant qu'arguments

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Regroupements

Tableaux en tant qu'objets (guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

En C#, les tableaux sont en fait des objets, et pas simplement des zones adressables de mémoire contiguë comme en C et C++. Array est le type de base abstrait de tous les types de tableau. Vous pouvez utiliser les propriétés et d'autres membres de classe qui Array possèdent. Par exemple, l'utilisation de la Length propriété pour récupérer la longueur d'un tableau. Le code suivant affecte la longueur du tableau `numbers`, c'est-à-dire la valeur `5`, à une variable appelée `lengthOfNumbers` :

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

La classe Array fournit beaucoup d'autres méthodes et propriétés utiles pour trier, rechercher et copier des tableaux.

Exemple

L'exemple suivant utilise la propriété Rank pour afficher le nombre de dimensions d'un tableau.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

Voir aussi

- Guide de programmation C#
- Tableaux
- Tableaux unidimensionnels
- Tableaux multidimensionnels
- Tableaux en escalier

Tableaux unidimensionnels (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Vous créez un tableau unidimensionnel à l'aide de l'opérateur New en spécifiant le type d'élément de tableau et le nombre d'éléments. L'exemple suivant déclare un tableau de cinq entiers :

```
int[] array = new int[5];
```

Ce tableau contient les éléments de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers.

Les tableaux peuvent stocker tout type d'élément que vous spécifiez, comme dans l'exemple suivant, qui déclare un tableau de chaînes :

```
string[] stringArray = new string[6];
```

Initialisation du tableau

Vous pouvez initialiser les éléments d'un tableau lorsque vous déclarez le tableau. Le spécificateur de longueur n'est pas nécessaire, car il est déduit par le nombre d'éléments dans la liste d'initialisation. Par exemple :

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

Le code suivant illustre une déclaration d'un tableau de chaînes où chaque élément de tableau est initialisé par un nom de jour :

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Vous pouvez éviter l' `new` expression et le type de tableau lorsque vous initialisez un tableau lors de la déclaration, comme indiqué dans le code suivant. C'est ce qu'on appelle un tableau implicitement typé:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Vous pouvez déclarer une variable tableau sans la créer, mais vous devez utiliser l' `new` opérateur quand vous assignez un nouveau tableau à cette variable. Par exemple :

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

Tableaux de types valeur et de types référence

Considérons la déclaration de tableau suivante :

```
SomeType[] array4 = new SomeType[10];
```

Le résultat de cette instruction varie selon que `SomeType` est un type valeur ou un type référence. S'il s'agit d'un type valeur, l'instruction crée un tableau de 10 éléments, chacun d'entre eux ayant le type `SomeType` . Si `SomeType` est un type référence, l'instruction crée un tableau de 10 éléments dont chacun est initialisé avec une référence null. Dans les deux instances, les éléments sont initialisés à la valeur par défaut pour le type d'élément. Pour plus d'informations sur les types valeur et les types référence, consultez types valeur et types référence.

Voir aussi

- Array
- Tableaux
- Tableaux multidimensionnels
- Tableaux en escalier

Tableaux multidimensionnels (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, la déclaration suivante crée un tableau à deux dimensions composé de quatre lignes et deux colonnes.

```
int[,] array = new int[4, 2];
```

La déclaration suivante crée un tableau de trois dimensions, 4, 2 et 3.

```
int[, ,] array1 = new int[4, 2, 3];
```

Initialisation du tableau

Vous pouvez initialiser le tableau au moment de la déclaration, comme le montre l'exemple suivant.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                       { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++) {
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

Vous pouvez également initialiser le tableau sans spécifier le rang.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Si vous choisissez de déclarer une variable de tableau sans initialisation, vous devez utiliser l'opérateur `new` pour assigner un tableau à la variable. L'utilisation de `new` est illustrée dans l'exemple suivant.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

L'exemple suivant assigne une valeur à un élément de tableau particulier.

```
array5[2, 1] = 25;
```

De la même manière, l'exemple suivant obtient la valeur d'un élément de tableau particulier et l'affecte à la

variable `elementValue`.

```
int elementValue = array5[2, 1];
```

L'exemple de code suivant initialise les éléments de tableau avec les valeurs par défaut (sauf pour les tableaux en escalier).

```
int[,] array6 = new int[10, 10];
```

Voir aussi

- Guide de programmation C#
- Tableaux
- Tableaux unidimensionnels
- Tableaux en escalier

Tableaux en escalier (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Un tableau en escalier est un tableau dont les éléments sont des tableaux. Les éléments d'un tableau en escalier peuvent être de dimensions et de tailles différentes. Un tableau en escalier est parfois appelé « tableau de tableaux ». Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier.

Voici une déclaration d'un tableau unidimensionnel qui comporte trois éléments, chacun d'eux étant un tableau unidimensionnel d'entiers :

```
int[][] jaggedArray = new int[3][];
```

Pour pouvoir utiliser `jaggedArray`, vous devez initialiser ses éléments. Pour ce faire, procédez comme suit :

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Chacun des éléments est un tableau unidimensionnel d'entiers. Le premier élément est un tableau de 5 entiers, le deuxième est un tableau de 4 entiers et le troisième est un tableau de 2 entiers.

Il est aussi possible d'utiliser des initialiseurs pour remplir les éléments de tableau de valeurs, auquel cas vous n'avez pas besoin de la taille du tableau. Par exemple :

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Vous pouvez aussi initialiser le tableau au moment de la déclaration, comme ceci :

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

Vous pouvez utiliser la forme abrégée suivante. Notez que vous ne pouvez pas omettre l'opérateur `new` dans l'initialisation des éléments, car il n'existe pas d'initialisation par défaut pour les éléments :

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

Un tableau en escalier est un tableau de tableaux, et par conséquent ses éléments sont des types référence et sont initialisés sur `null`.

Vous pouvez accéder aux éléments de tableau individuels comme dans ces exemples :

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Vous trouverez ci-dessous une déclaration et une initialisation d'un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes. Pour plus d'informations sur les tableaux à deux dimensions, consultez Tableaux multidimensionnels.

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

Vous pouvez accéder aux différents éléments comme le montre cet exemple, qui présente la valeur de l'élément `[1,0]` du premier tableau (valeur `5`) :

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

La méthode `Length` retourne le nombre de tableaux contenus dans le tableau en escalier. Par exemple, en supposant que vous avez déclaré le tableau précédent, cette ligne :

```
System.Console.WriteLine(jaggedArray4.Length);
```

retourne la valeur 3.

Exemple

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Voir aussi

- Array
- Guide de programmation C#
- Tableaux
- Tableaux unidimensionnels
- Tableaux multidimensionnels

Utiliser foreach avec des tableaux (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction foreach offre une méthode simple et appropriée pour itérer au sein des éléments d'un tableau.

Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1` :

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite vers la gauche :

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités.

Voir aussi

- Array
- Guide de programmation C#
- Tableaux
- Tableaux unidimensionnels
- Tableaux multidimensionnels
- Tableaux en escalier

Passage de tableaux en tant qu'arguments (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Les tableaux peuvent être passés en tant qu'arguments à des paramètres de méthode. Comme les tableaux sont des types référence, la méthode peut modifier la valeur des éléments.

Passage de tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Par exemple, l'instruction suivante envoie un tableau à une méthode Print.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

Le code suivant illustre une implémentation partielle de la méthode Print.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Exemple

Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Passage de tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant :

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Exemple

Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Element(0,0)=1
        Element(0,1)=2
        Element(1,0)=3
        Element(1,1)=4
        Element(2,0)=5
        Element(2,1)=6
        Element(3,0)=7
        Element(3,1)=8
    */
```

Voir aussi

- Guide de programmation C#
- Tableaux
- Tableaux unidimensionnels
- Tableaux multidimensionnels
- Tableaux en escalier

Tableaux implicitement typés (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les tableaux implicitement typés sont généralement utilisés dans les expressions de requête avec des types anonymes et des initialiseurs d'objets et de collections.

Les exemples suivants montrent comment créer un tableau implicitement typé :

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation. Notez aussi que les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Tableaux implicitement typés dans les initialiseurs d'objets

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Notez que le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Voir aussi

- Guide de programmation C#
- Variables locales implicitement typées
- Tableaux
- Types anonymes
- Initialiseurs d'objets et de collections
- var
- LINQ en C#

Chaînes (Guide de programmation C#)

02/11/2020 • 22 minutes to read • Edit Online

Une chaîne est un objet de type String dont la valeur est du texte. En interne, le texte est stocké sous la forme d'une collection séquentielle en lecture seule d'objets Char. Il n'existe aucun caractère de fin Null à la fin d'une chaîne C# ; par conséquent, une chaîne C# peut contenir n'importe quel nombre de caractères Null incorporés ('\0'). La propriété Length d'une chaîne représente le nombre d'objets `Char` qu'elle contient, et non pas le nombre de caractères Unicode. Pour accéder à des points de code Unicode individuels dans une chaîne, utilisez l'objet StringInfo.

String et System. String

En C#, le mot clé `string` est un alias pour String. Par conséquent, `String` et `string` sont équivalents et vous pouvez utiliser la convention d'affectation de noms que vous préférez. La classe `String` fournit de nombreuses méthodes pour créer, manipuler et comparer des chaînes en toute sécurité. En outre, le langage C# surcharge certains opérateurs pour simplifier les opérations de chaînes courantes. Pour plus d'informations sur le mot clé, voir chaîne. Pour plus d'informations sur le type et ses méthodes, consultez String.

Déclaration et initialisation de chaînes

Vous pouvez déclarer et initialiser des chaînes de différentes manières, comme illustré dans l'exemple suivant :

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Notez que vous n'utilisez pas l'opérateur new pour créer un objet String, excepté lors de l'initialisation de la

chaîne avec un tableau de caractères.

Initialisez une chaîne avec la valeur constante Empty pour créer un objet String dont la chaîne est de longueur nulle. La représentation de littéral de chaîne d'une chaîne de longueur nulle est "". En initialisant les chaînes avec la valeur Empty au lieu de null, vous pouvez réduire les risques de levée de l'exception NullReferenceException. Utilisez la méthode statique IsNullOrEmpty(String) pour vérifier la valeur d'une chaîne avant de tenter d'y accéder.

Immuabilité des objets String

Les objets String sont *immuables* : ils ne peuvent pas être modifiés une fois qu'ils ont été créés. Toutes les méthodes String et tous les opérateurs C# qui semblent modifier une chaîne retournent en fait les résultats dans un nouvel objet string. Dans l'exemple suivant, lorsque les contenus de `s1` et `s2` sont concaténés pour former une chaîne unique, les deux chaînes d'origine restent inchangées. L'opérateur `+=` crée une nouvelle chaîne qui contient le contenu combiné. Ce nouvel objet est assigné à la variable `s1` et l'objet d'origine qui a été assigné à `s1` est libéré pour la garbage collection, car aucune autre variable ne le référence.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Étant donné qu'une « modification » de chaîne correspond en fait à la création d'une nouvelle chaîne, vous devez être prudent lorsque vous créez des références aux chaînes. Si vous créez une référence à une chaîne, puis « modifiez » la chaîne d'origine, la référence continue à pointer vers l'objet d'origine au lieu du nouvel objet créé lors de la modification de la chaîne. Le code suivant illustre ce comportement :

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

Pour plus d'informations sur la création de chaînes basées sur des modifications telles que les opérations de recherche et de remplacement sur la chaîne d'origine, consultez Comment modifier le contenud'une chaîne.

Littéraux de chaînes normales et textuelles

Utilisez des littéraux de chaînes normales lorsque vous devez incorporer des caractères d'échappement fournis par C#, comme illustré dans l'exemple suivant :

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Utilisez des chaînes textuelles pour plus de commodité et pour une meilleure lisibilité lorsque le texte de la chaîne contient des barres obliques inverses, par exemple dans les chemins d'accès. Étant donné que les chaînes textuelles conservent les caractères de nouvelle ligne dans le texte de la chaîne, elles peuvent servir à initialiser des chaînes multilignes. Utilisez des guillemets doubles pour incorporer des guillemets dans une chaîne textuelle. L'exemple suivant montre certaines utilisations courantes des chaînes textuelles :

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Séquences d'échappement de chaîne

SÉQUENCE D'ÉCHAPPEMENT	NOM DU CARACTÈRE	CODAGE UNICODE
\'	Guillemet simple	0x0027
\"	Guillemet double	0x0022
\\	Barre oblique inverse	0x005C
\0	Null	0x0000
\a	Alerte	0x0007
\b	Retour arrière	0x0008
\f	Saut de page	0x000C
\n	Nouvelle ligne	0x000A
\r	Retour chariot	0x000D

SÉQUENCE D'ÉCHAPPEMENT	NOM DU CARACTÈRE	CODAGE UNICODE
\t	Tabulation horizontale	0x0009
\v	Tabulation verticale	0x000B
\u	Séquence d'échappement Unicode (UTF-16)	`\uHHHH` (plage : 0000-FFFF ; exemple : `\u00E7` = "ç")
\U	Séquence d'échappement Unicode (UTF-32)	`\U00HHHHHH` (plage : 000000-10FFFF ; exemple : `\U0001F47D` = "& # x1F47D ;")
\x	Séquence d'échappement Unicode similaire à "\u", mais avec une longueur variable	`\xH[H][H][H]` (plage : 0-FFFF ; exemple : `\x00E7` or `\x0E7` ou `\xE7` = "ç")

WARNING

Quand vous utilisez la séquence d'échappement `\x` et spécifiez moins de 4 chiffres hexadécimaux, si les caractères qui suivent immédiatement la séquence d'échappement sont des chiffres hexadécimaux valides (par ex. 0-9, A-F et a-f), ils sont interprétés comme faisant partie de la séquence d'échappement. Par exemple, `\xA1` donne "¡", qui est le point de code U+00A1. Toutefois, si le caractère suivant est « A » ou « a », la séquence d'échappement est plutôt être interprétée comme étant `\xA1A` et donne "■", qui est le point de code U+0A1A. Dans ce cas, la spécification des 4 chiffres hexadécimaux (par ex. `\x00A1`) empêche toute mauvaise interprétation possible.

NOTE

Au moment de la compilation, les chaînes textuelles sont converties en chaînes normales avec les mêmes séquences d'échappement. Par conséquent, si vous affichez une chaîne textuelle dans la fenêtre Espion du débogueur, vous verrez les caractères d'échappement qui ont été ajoutés par le compilateur et non la version textuelle de votre code source. Par exemple, la chaîne textuelle `@"C:\files.txt"` s'affiche dans la fenêtre Espion en tant que "C:\\files.txt".

Chaînes de format

Une chaîne de format est une chaîne dont le contenu est déterminé de façon dynamique lors de l'exécution. Les chaînes de format sont créées en incorporant des *expressions interpolées* ou des espaces réservés à l'intérieur d'accolades dans une chaîne. Tous les éléments à l'intérieur des accolades (`{...}`) seront convertis en une valeur et affichés sous forme d'une chaîne mise en forme lors de l'exécution. Il existe deux méthodes pour créer des chaînes de format : l'interpolation de chaîne et la mise en forme composite.

Interpolation de chaîne

Disponible dans C# 6.0 et versions ultérieures, les *chaînes interpolées* sont identifiées par le caractère spéciale `$` et incluent des expressions interpolées entre accolades. Si vous ne connaissez pas l'interpolation de chaîne, consultez le tutoriel interactif Interpolation de chaînes en C# pour obtenir un aperçu.

Utilisez l'interpolation de chaîne pour améliorer la lisibilité et la maintenance de votre code. L'interpolation de chaîne permet d'obtenir les mêmes résultats que la méthode `String.Format`, mais avec plus de facilité d'utilisation et de clarté.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Mise en forme composite

String.Format utilise des espaces réservés entre accolades pour créer une chaîne de format. Cet exemple renvoie une sortie similaire à la méthode d'interpolation de chaîne utilisée ci-dessus.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Pour plus d'informations sur la mise en forme des types .NET, consultez Mise en forme des types dans .NET.

Sous-chaînes

Une sous-chaîne est une séquence de caractères qui est contenue dans une chaîne. Utilisez la méthode Substring pour créer une chaîne à partir d'une partie de la chaîne d'origine. Vous pouvez rechercher une ou plusieurs occurrences d'une sous-chaîne en utilisant la méthode IndexOf. Utilisez la méthode Replace pour remplacer toutes les occurrences d'une sous-chaîne spécifiée par une nouvelle chaîne. Comme la méthode Substring, Replace retourne en fait une nouvelle chaîne et ne modifie pas la chaîne d'origine. Pour plus d'informations, consultez Comment rechercher des chaînes et Comment modifier le contenu d'une chaîne.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accès aux caractères individuels

Vous pouvez utiliser la notation de tableau avec une valeur d'index pour obtenir un accès en lecture seule aux caractères individuels, comme dans l'exemple suivant :

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Si les méthodesString ne fournissent pas la fonctionnalité nécessaire pour modifier les caractères individuels d'une chaîne, vous pouvez utiliser un objet StringBuilder pour modifier les caractères individuels « sur place », puis créer une chaîne pour stocker les résultats en utilisant les méthodes StringBuilder. Dans l'exemple suivant, supposons que vous devez modifier la chaîne d'origine d'une façon particulière, puis stocker les résultats pour une utilisation ultérieure :

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Chaînes Null et chaînes vides

Une chaîne vide est une instance d'un objet System.String qui contient zéro caractère. Les chaînes vides sont souvent utilisées dans divers scénarios de programmation pour représenter un champ de texte vide. Vous pouvez appeler des méthodes sur des chaînes vides, car ce sont des objets System.String. Les chaînes vides sont initialisées comme suit :

```
string s = String.Empty;
```

En revanche, une chaîne null ne fait pas référence à une instance d'un objet System.String, et toute tentative pour appeler une méthode sur une chaîne null lève une exception NullReferenceException. Toutefois, vous pouvez utiliser des chaînes Null dans les opérations de comparaison et de concaténation avec d'autres chaînes. Les exemples suivants illustrent certains cas dans lesquels une référence à une chaîne Null provoque ou non la levée d'une exception :

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Utilisation de StringBuilder pour créer rapidement une chaîne

Les opérations Strings dans .NET sont hautement optimisées et, dans la plupart des cas, n'ont pas d'incidence significative sur les performances. Toutefois, dans certains scénarios, notamment dans le cas de boucles serrées qui s'exécutent plusieurs centaines voire milliers de fois, les opérations String peuvent affecter les performances. La classe StringBuilder crée un tampon de chaîne qui offre de meilleures performances si votre programme exécute de nombreuses manipulations de chaînes. La chaîne StringBuilder vous permet également de réassigner des caractères individuels, une fonctionnalité non prise en charge par les types de données string intégrés. Ce code, par exemple, modifie le contenu d'une chaîne sans créer de nouvelle chaîne :

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

Dans cet exemple, un objet StringBuilder est utilisé pour créer une chaîne à partir d'un ensemble de types numériques :

```
using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            var sb = new StringBuilder();

            // Create a string composed of numbers 0 - 9
            for (int i = 0; i < 10; i++)
            {
                sb.Append(i.ToString());
            }
            Console.WriteLine(sb);  // displays 0123456789

            // Copy one character of the string (not possible with a System.String)
            sb[0] = sb[9];

            Console.WriteLine(sb);  // displays 9123456789
            Console.WriteLine();
        }
    }
}
```

Chaînes, méthodes d'extension et LINQ

Étant donné que le type String implémente IEnumerable<T>, vous pouvez utiliser les méthodes d'extension définies dans la classe Enumerable sur des chaînes. Pour éviter une surcharge visuelle, ces méthodes sont exclues d'IntelliSense pour le type String, mais elles restent néanmoins disponibles. Vous pouvez également utiliser des expressions de requête LINQ sur des chaînes. Pour plus d'informations, consultez LINQ et Strings.

Rubriques connexes

RUBRIQUE	DESCRIPTION
Comment modifier le contenu d'une chaîne	Illustre les techniques permettant de transformer des chaînes et de modifier le contenu de chaînes.
Comment comparer des chaînes	Montre comment effectuer des comparaisons de chaînes ordinales et spécifiques à la culture.
Concaténation de plusieurs chaînes	Illustre différentes manières de joindre plusieurs chaînes pour en former une.
Comment analyser des chaînes à l'aide de String. Split	Contient un exemple de code qui illustre l'utilisation de la méthode `String.Split` pour analyser des chaînes.
Comment rechercher des chaînes	Explique comment rechercher du texte ou des modèles spécifiques dans des chaînes.
Comment déterminer si une chaîne représente une valeur numérique	Montre comment analyser une chaîne en toute sécurité pour déterminer si elle possède une valeur numérique valide.
Interpolation de chaîne	Décrit la fonctionnalité d'interpolation de chaîne qui fournit une syntaxe pratique pour les chaînes de format.

RUBRIQUE	DESCRIPTION
Opérations de chaînes de base	Fournit des liens vers des rubriques utilisant les méthodes System.String et System.Text.StringBuilder pour effectuer des opérations de chaînes de base.
Analyse de chaînes	Décrit comment convertir des représentations sous forme de chaîne de types de base .NET en instances de types correspondants.
Analyse des chaînes de date et d'heure dans .NET	Montre comment convertir une chaîne telle que « 24/01/2008 » en objet System.DateTime.
Comparaison de chaînes	Inclut des informations sur la façon de comparer des chaînes et fournit des exemples en C# et Visual Basic.
Utilisation de la classe StringBuilder	Explique comment créer et modifier des objets string dynamiques avec la classe StringBuilder.
LINQ et chaînes	Fournit des informations sur l'exécution de différentes opérations sur des chaînes à l'aide de requêtes LINQ.
Guide de programmation C#	Fournit des liens vers des rubriques qui expliquent les constructions de programmation en C#.

Comment déterminer si une chaîne représente une valeur numérique (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Pour déterminer si une chaîne est une représentation valide d'un type numérique spécifié, utilisez la méthode statique `TryParse` implémentée par tous les types numériques primitifs et par les types tels que DateTime et IPAddress. L'exemple suivant montre comment déterminer si « 108 » est une chaîne int valide.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Si la chaîne contient des caractères non numériques ou si la valeur numérique est trop grande ou trop petite pour le type particulier que vous avez spécifié, `TryParse` retourne la valeur false et affecte la valeur zéro comme paramètre de sortie. Sinon, elle retourne la valeur true et affecte la valeur numérique de la chaîne comme paramètre de sortie.

> **NOTE**
>
> Une chaîne peut contenir uniquement des caractères numériques et ne pas être valide pour le type dont vous utilisez la méthode `TryParse`. Par exemple, « 256 » n'est pas une valeur valide pour `byte`, mais elle est valide pour `int`. « 98,6 » n'est pas une valeur valide pour `int`, mais elle est valide pour `decimal`.

Exemple

Les exemples suivants montrent comment utiliser `TryParse` avec des représentations sous forme de chaîne de valeurs `long`, `byte` et `decimal`.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

Programmation fiable

Les types numériques primitifs implémentent également la méthode statique `Parse`, qui lève une exception si la chaîne n'est pas un nombre valide. `TryParse` est généralement plus efficace, car elle retourne simplement false si le nombre n'est pas valide.

Sécurité .NET

Utilisez toujours la méthode `TryParse` ou `Parse` pour valider l'entrée utilisateur à partir de contrôles tels que des zones de texte et des zones de liste déroulante.

Voir aussi

- Comment convertir un tableau d'octets en entier
- Comment convertir une chaîne en nombre
- Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques
- Analyse des chaînes numériques
- Mise en forme des types

Indexeurs (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Les indexeurs permettent aux instances d'une classe ou d'un struct d'être indexés comme des tableaux. La valeur indexée peut être définie ou récupérée sans spécifier explicitement un membre de type ou d'instance. Les indexeurs s'apparentent aux propriétés à l'exception près que leurs accesseurs acceptent des paramètres.

L'exemple suivant définit une classe générique avec des méthodes d'accesseur get et set simples pour attribuer et récupérer des valeurs. La classe `Program` classe crée une instance de cette classe pour le stockage des chaînes.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

Pour plus d'exemples, consultez Rubriques connexes.

Définitions de corps d'expression

Il est courant pour l'accesseur get ou set d'un indexeur d'être constitué d'une instruction unique qui retourne ou définit une valeur. Les membres expression-bodied fournissent une syntaxe simplifiée pour prendre en charge ce scénario. À partir de C# 6, un indexeur en lecture seule peut être implémenté comme un membre expression-bodied, comme le montre l'exemple suivant.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Notez que `=>` introduit le corps de l'expression et que le mot clé `get` n'est pas utilisé.

À partir de C# 7.0, l'accesseur get et l'accesseur set peuvent être implémentés en tant que membres expression-bodied. Dans ce cas, les deux mots clés `get` et `set` doivent être utilisés. Par exemple :

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Vue d'ensemble des indexeurs

- Les indexeurs permettent aux objets d'être indexés d'une manière similaire aux tableaux.
- Un accesseur `get` retourne une valeur. Un accesseur `set` affecte une valeur.
- Le mot clé this est utilisé pour définir l'indexeur.
- Le mot clé value est utilisé pour définir la valeur assignée par l'accesseur `set`.
- Les indexeurs n'ont pas besoin d'être indexés par une valeur entière. Il vous appartient de choisir comment définir le mécanisme de recherche spécifique.
- Les indexeurs peuvent être surchargés.
- Les indexeurs peuvent avoir plusieurs paramètres formels, par exemple, quand vous accédez à un tableau à deux dimensions.

Sections connexes

- Utilisation d'indexeurs
- Indexeurs dans les interfaces
- Comparaison entre propriétés et indexeurs
- Restriction d'accessibilité de l'accesseur

Spécification du langage C#

Pour plus d'informations, consultez Indexeurs dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Propriétés

Utiliser des indexeurs (Guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Les indexeurs sont une pratique syntaxique qui vous permet de créer une classe, un structou une interface que les applications clientes peuvent accéder en tant que tableau. Le compilateur génère une `Item` propriété (ou une propriété autre que nommée si IndexerNameAttribute est présent) et les méthodes d'accesseur appropriées. Le plus souvent, les indexeurs sont implémentés dans les types dont l'objectif premier est d'encapsuler une collection ou un tableau interne. Par exemple, supposons que vous ayez une classe `TempRecord` qui représente la température en degrés Fahrenheit enregistrée à 10 heures différentes pendant une période de 24 heures. La classe contient un `temps` tableau de type `float[]` pour stocker les valeurs de température. En implémentant un indexeur dans cette classe, les clients peuvent accéder aux températures dans une instance `TempRecord` sous la forme `float temp = tempRecord[4]` et non sous la forme `float temp = tempRecord.temps[4]`. La notation de l'indexeur simplifie non seulement la syntaxe des applications clientes ; elle rend également la classe et son objectif plus intuitifs à comprendre pour d'autres développeurs.

Pour déclarer un indexeur sur une classe ou un struct, utilisez le mot clé this, comme dans l'exemple suivant :

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

La déclaration d'un indexeur génère automatiquement une propriété nommée `Item` sur l'objet. La `Item` propriété n'est pas directement accessible à partir de l' expression d'accès au membred'instance. En outre, si vous ajoutez votre propre `Item` propriété à un objet avec un indexeur, vous obtiendrez une Erreur du compilateur CS0102. Pour éviter cette erreur, utilisez l' IndexerNameAttribute indexeur renommez-le comme indiqué ci-dessous.

Notes

Le type d'un indexeur et le type de ses paramètres doivent être au moins aussi accessibles que l'indexeur lui-même. Pour plus d'informations sur les niveaux d'accessibilité, consultez Modificateurs d'accès.

Pour plus d'informations sur l'utilisation d'indexeurs avec une interface, consultez Indexeurs d'interface.

La signature d'un indexeur est composée du nombre et des types de ses paramètres formels. Elle ne comporte ni le type de l'indexeur ni le nom des paramètres formels. Si vous déclarez plusieurs indexeurs dans la même classe, ils doivent avoir des signatures différentes.

Une valeur d'indexeur n'est pas classée comme variable ; vous ne pouvez donc pas la passer comme paramètre ref ou out.

Pour affecter à l'indexeur un nom exploitable dans d'autres langages, utilisez System.Runtime.CompilerServices.IndexerNameAttribute, comme dans l'exemple suivant :

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Cet indexeur porte le nom `TheItem` , car il est substitué par l'attribut de nom de l'indexeur. Par défaut, le nom de l'indexeur est `Item` .

Exemple 1

L'exemple suivant montre comment déclarer un champ de tableau privé `temps`, et un indexeur. L'indexeur permet d'accéder directement à l'instance `tempRecord[i]`. Comme alternative à l'utilisation de l'indexeur, vous pouvez déclarer le tableau comme membre public et accéder directement à ses membres, `tempRecord.temps[i]`.

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Notez que quand l'accès à un indexeur est évalué, par exemple dans une instruction `Console.Write`, l'accesseur get est appelé. C'est pourquoi une erreur de compilation se produit s'il n'existe aucun accesseur `get`.

```
using System;

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
}
```

Indexation avec d'autres valeurs

C# ne limite pas le type de paramètre d'indexeur au type entier. Par exemple, il peut être utile d'utiliser une chaîne avec un indexeur. Il est possible d'implémenter un tel indexeur en recherchant la chaîne dans la collection et en retournant la valeur appropriée. Comme les accesseurs peuvent être surchargés, les versions de chaîne et d'entier peuvent coexister.

Exemple 2

L'exemple suivant déclare une classe qui stocke les jours de la semaine. Un accesseur `get` prend une chaîne, le nom d'un jour, et retourne l'entier correspondant. Par exemple, « Sunday » retourne 0, « Monday » retourne 1 et ainsi de suite.

```
using System;

// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form \"Sun\", \"Mon\", etc");
    }
}
```

Utilisation de l'exemple 2

```
using System;

class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
        {
            Console.WriteLine(week["Made-up day"]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported.
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}
```

Exemple 3

L'exemple suivant déclare une classe qui stocke les jours de la semaine à l'aide de l' System.DayOfWeek énumération. Un `get` accesseur prend un `DayOfWeek` , la valeur d'un jour et retourne l'entier correspondant. Par exemple, `DayOfWeek.Sunday` retourne 0, `DayOfWeek.Monday` retourne 1, et ainsi de suite.

```
using System;
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined System.DayOfWeek value.");
    }
}
```

Utilisation de l'exemple 3

```
using System;

class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}
```

Programmation fiable

La sécurité et la fiabilité des indexeurs peuvent être améliorées de deux manières principales :

- N'oubliez pas d'incorporer une stratégie de gestion des erreurs au cas où le code client passerait une valeur d'index non valide. Dans le premier exemple décrit plus haut dans cette rubrique, la classe TempRecord fournit une propriété Length qui permet au code client de vérifier l'entrée avant de la passer à l'indexeur. Vous pouvez également placer le code de gestion des erreurs à l'intérieur de l'indexeur lui-même. N'oubliez

pas d'indiquer aux utilisateurs toutes les exceptions que vous levez dans un accesseur d'indexeur.

- Définissez pour les accesseurs get et set une accessibilité aussi restrictive que possible. Cela est particulièrement important dans le cas de l'accesseur `set`. Pour plus d'informations, consultez Restriction d'accessibilité de l'accesseur.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés

Indexeurs dans les interfaces (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Des indexeurs peuvent être déclarés dans une interface. Les accesseurs d'indexeurs d'interface se distinguent sur plusieurs plans des accesseurs d'indexeurs de classe, à savoir :

- Les accesseurs d'interface n'utilisent pas de modificateurs.
- Un accesseur d'interface n'a généralement pas de corps.

L'objectif de l'accesseur est d'indiquer si l'indexeur est en lecture-écriture, en lecture seule ou en écriture seule. Vous pouvez fournir une implémentation pour un indexeur défini dans une interface, mais cela est rare. Les indexeurs définissent généralement une API pour accéder aux champs de données, et les champs de données ne peuvent pas être définis dans une interface.

L'exemple ci-dessous porte sur un accesseur d'indexeur d'interface :

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

La signature d'un indexeur doit se distinguer de tous les autres indexeurs déclarés dans la même interface.

Exemple

L'exemple suivant montre comment implémenter des indexeurs d'interface.

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```
IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

Dans l'exemple précédent, vous pouvez utiliser l'implémentation de membre d'interface explicite en utilisant le nom qualifié complet du membre d'interface. Par exemple

```
string IIndexInterface.this[int index]
{
}
```

Cependant, le nom qualifié complet est seulement nécessaire pour éviter toute ambiguïté quand la classe implémente plusieurs interfaces avec la même signature d'indexeur. Par exemple, si une classe `Employee` implémente deux interfaces, `ICitizen` et `IEmployee`, et que les deux interfaces ont la même signature d'indexeur, l'implémentation de membre d'interface explicite est nécessaire. Autrement dit, la déclaration d'indexeur suivante :

```
string IEmployee.this[int index]
{
}
```

implémente l'indexeur dans l'interface `IEmployee`, alors que la déclaration suivante :

```
string ICitizen.this[int index]
{
}
```

implémente l'interface dans l'interface `ICitizen`.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés
- Interfaces

Comparaison entre propriétés et indexeurs (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Les indexeurs sont semblables aux propriétés. À l'exception des différences répertoriées dans le tableau suivant, toutes les règles définies pour les accesseurs des propriétés s'appliquent également aux accesseurs des indexeurs.

PROPRIÉTÉ	INDEXATION
Permet aux méthodes d'être appelées comme si elles étaient des membres de données publics.	Permet aux éléments d'une collection interne d'un objet d'être accessibles à l'aide de la notation de tableau sur l'objet lui-même.
Accessible par le biais d'un nom simple.	Accessible par le biais d'un index.
Peut être un membre statique ou un membre d'instance.	Doit être un membre d'instance.
Un accesseur get d'une propriété n'a aucun paramètre.	Un accesseur `get` d'un indexeur possède la même liste de paramètres formels que l'indexeur.
Un accesseur set d'une propriété contient le paramètre `value` implicite.	Un accesseur `set` d'un indexeur possède la même liste de paramètres formels que l'indexeur, outre le paramètre value.
Prend en charge la syntaxe abrégée avec les propriétés implémentées automatiquement.	Prend en charge les membres expression-bodied pour les indexeurs get-only.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés

Événements (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Les événements permettent à une classe ou à un objet de notifier d'autres classes ou objets quand quelque chose de significatif se produit. La classe qui envoie (ou *déclenche*) l'événement est appelée *publieur* et les classes qui reçoivent (ou *gèrent*) l'événement sont appelées *abonnés*.

Dans une application C# Windows Forms ou web classique, vous vous abonnez à des événements déclenchés par des contrôles, comme des boutons et des zones de liste. Vous pouvez utiliser l'IDE Visual C# pour parcourir les événements publiés par un contrôle et sélectionner ceux que vous voulez gérer. L'IDE permet d'ajouter automatiquement une méthode de gestionnaire d'événements vide et le code pour vous abonner à l'événement. Pour plus d'informations, consultez Comment s'abonner et annuler l'abonnement à des événements.

Vue d'ensemble des événements

Les événements ont les propriétés suivantes :

- Le publieur détermine quand un événement est déclenché ; les abonnés déterminent l'action entreprise en réponse à l'événement.
- Un événement peut avoir plusieurs abonnés. Un abonné peut gérer plusieurs événements provenant de plusieurs publieurs.
- Les événements qui n'ont aucun abonné ne sont jamais déclenchés.
- Les événements sont généralement utilisés pour signaler des actions de l'utilisateur, comme les clics de bouton ou les sélections de menu dans les interfaces utilisateur graphiques.
- Quand un événement a plusieurs abonnés, les gestionnaires d'événements sont appelées de façon synchrone quand un événement est déclenché. Pour appeler des événements de façon asynchrone, consultez Calling Synchronous Methods Asynchronously.
- Dans la bibliothèque de classes .NET, les événements sont basés sur le EventHandler délégué et la EventArgs classe de base.

Sections connexes

Pour plus d'informations, consultez les pages suivantes :

- Comment s'abonner à des événements et s'en désabonner
- Comment publier des événements conformes aux indications de .NET
- Comment déclencher des événements de la classe de base dans les classes dérivées
- Comment implémenter des événements d'interface
- Comment implémenter des accesseurs d'événement personnalisés

Spécification du langage C#

Pour plus d'informations, consultez Événements dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Chapitres proposés

Delegates, Events, and Lambda Expressions (Délégués, événements et expressions lambda) dans C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events dans Learning C# 3.0: Master the fundamentals of C# 3.0

Voir aussi

- EventHandler
- Guide de programmation C#
- Délégués
- Création de gestionnaires d'événements dans les Windows Forms

Comment s'abonner et annuler l'abonnement à des événements (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Vous vous abonnez à un événement publié par une autre classe lorsque vous voulez écrire du code personnalisé qui doit être appelé quand cet événement est déclenché. Par exemple, vous pouvez vous abonner à l'événement `click` d'un bouton pour permettre à votre application de réagir lorsque l'utilisateur clique sur le bouton.

Pour s'abonner aux événements à l'aide de l'IDE de Visual Studio

1. Si vous ne voyez pas la fenêtre **Propriétés**, en mode **Création**, cliquez sur le formulaire ou le contrôle pour lequel vous voulez créer un gestionnaire d'événements, puis sélectionnez **Propriétés**.

2. En haut de la fenêtre **Propriétés**, cliquez sur l'icône **Événements**.

3. Double-cliquez sur l'événement que vous voulez créer, par exemple l'événement `Load`.

   Visual C# crée une méthode de gestionnaire d'événements vide et l'ajoute à votre code. Vous pouvez également ajouter le code manuellement en mode **Code**. Par exemple, les lignes de code suivantes déclarent une méthode de gestionnaire d'événements qui est appelée lorsque la classe `Form` déclenche l'événement `Load`.

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   La ligne de code qui est nécessaire pour s'abonner à l'événement est aussi générée automatiquement dans la méthode `InitializeComponent`, dans le fichier Form1.Designer.cs de votre projet. Elle ressemble à ceci :

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

Pour s'abonner aux événements par programmation

1. Définissez une méthode de gestionnaire d'événements dont la signature correspond à la signature du délégué de l'événement. Par exemple, si l'événement est basé sur le type délégué EventHandler, le code suivant représente le stub de méthode :

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Utilisez l'opérateur d'affectation d'addition (`+=`) pour attacher un gestionnaire d'événements à l'événement. Dans l'exemple suivant, nous allons supposer qu'un objet nommé `publisher` a un événement nommé `RaiseCustomEvent`. Notez que la classe d'abonné nécessite une référence à la classe d'éditeur pour s'abonner à ses événements.

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

Notez que la syntaxe précédente est une nouveauté du langage C# 2.0. Elle équivaut exactement à la syntaxe du C# 1.0, dans laquelle le délégué d'encapsulation doit être explicitement créé à l'aide du mot clé `new` :

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

Vous pouvez également utiliser une expression lambda pour spécifier un gestionnaire d'événements :

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
        {
            MessageBox.Show(((MouseEventArgs)e).Location.ToString());
        };
}
```

Pour s'abonner aux événements à l'aide d'une méthode anonyme

- Si vous savez que vous n'aurez pas à vous désabonner d'un événement, vous pouvez utiliser l'opérateur d'assignation d'addition (`+=`) pour attacher une méthode anonyme à l'événement. Dans l'exemple suivant, nous supposons qu'un objet nommé `publisher` a un événement nommé `RaiseCustomEvent`, et qu'une classe `CustomEventArgs` a également été définie pour contenir des informations d'événements spécialisés. Notez que la classe d'abonné nécessite une référence à `publisher` pour s'abonner à ses événements.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Il est important de noter que vous ne pourrez pas vous désabonner facilement d'un événement si vous avez utilisé une fonction anonyme pour vous y inscrire. Pour vous désinscrire dans ce scénario, accédez au code dans lequel vous vous êtes abonné à l'événement, stockez la méthode anonyme dans une variable de délégué, puis ajoutez le délégué à l'événement. En général, nous recommandons de ne pas utiliser de fonctions anonymes pour vous abonner aux événements si vous devez vous en désabonner plus tard dans votre code. Pour plus d'informations sur les fonctions anonymes, consultez Fonctions anonymes.

Désabonnement

Pour éviter que votre gestionnaire d'événements ne soit appelé lorsque l'événement est déclenché, désabonnez-vous de l'événement. Pour empêcher les fuites de ressources, vous devez vous désabonner des événements avant d'éliminer un objet d'abonné. Tant que vous êtes abonné à un événement, le délégué de multidiffusion qui se trouve sous l'événement, dans l'objet de publication, comporte une référence au délégué qui encapsule le gestionnaire d'événements de l'abonné. Tant que l'objet de publication contient cette référence, le garbage collection ne supprime pas l'objet d'abonné.

Pour se désabonner d'un événement

- Utilisez l'opérateur d'assignation de soustraction (`-=`) pour vous désabonner d'un événement :

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

Lorsque tous les abonnés se sont désabonnés d'un événement, l'instance d'événement de la classe

d'éditeur est définie sur `null`.

Voir aussi

- Événements
- event
- Comment publier des événements conformes aux indications de .NET
- opérateurs-and-=
- opérateurs + et + =

Comment publier des événements conformes aux instructions .NET (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

La procédure suivante montre comment ajouter des événements qui suivent le modèle .NET standard à vos classes et structures. Tous les événements de la bibliothèque de classes .NET sont basés sur le EventHandler délégué, qui est défini comme suit :

```
public delegate void EventHandler(object sender, EventArgs e);
```

> **NOTE**
> .NET Framework 2,0 introduit une version générique de ce délégué, EventHandler<TEventArgs> . Les exemples suivants montrent comment utiliser les deux versions.

Bien que les événements dans les classes que vous définissez puissent être basés sur n'importe quel type délégué valide, même les délégués qui retournent une valeur, il est généralement recommandé de baser vos événements sur le modèle .NET à l'aide de EventHandler , comme indiqué dans l'exemple suivant.

Le nom `EventHandler` peut entraîner un peu de confusion, car il ne gère pas réellement l'événement. Le EventHandler générique et EventHandler<TEventArgs> sont des types délégués. Une méthode ou une expression lambda dont la signature correspond à la définition de délégué est le *Gestionnaire d'événements* et sera appelée lorsque l'événement est déclenché.

Publier des événements basés sur le modèle EventHandler

1. (Ignorez cette étape et passez à l'étape 3A si vous n'avez pas besoin d'envoyer des données personnalisées avec votre événement.) Déclarez la classe pour vos données personnalisées au niveau d'une étendue qui est visible à la fois pour vos classes de serveur de publication et d'abonné. Ajoutez ensuite les membres nécessaires pour contenir vos données d'événement personnalisées. Dans cet exemple, une simple chaîne est retournée.

   ```
   public class CustomEventArgs : EventArgs
   {
       public CustomEventArgs(string message)
       {
           Message = message;
       }

       public string Message { get; set; }
   }
   ```

2. (Ignorez cette étape si vous utilisez la version générique de EventHandler<TEventArgs> .) Déclarez un délégué dans votre classe de publication. Donnez-lui un nom qui se termine par `EventHandler` . Le deuxième paramètre spécifie votre `EventArgs` type personnalisé.

   ```
   public delegate void CustomEventHandler(object sender, CustomEventArgs args);
   ```

3. Déclarez l'événement dans votre classe de publication en effectuant l'une des étapes suivantes.

   a. Si vous n'avez aucune classe EventArgs personnalisée, votre type d'événement sera le délégué EventHandler non générique. Il est inutile de déclarer le délégué, car il l'est déjà dans l'espace de noms System inclus au moment où vous créez votre projet C#. Ajoutez le code suivant à votre classe de serveur de publication.

   ```
   public event EventHandler RaiseCustomEvent;
   ```

   b. Si vous utilisez la version non générique de EventHandler et que vous avez une classe personnalisée dérivée de EventArgs, déclarez votre événement à l'intérieur de votre classe de publication et utilisez votre délégué de l'étape 2 comme type.

   ```
   public event CustomEventHandler RaiseCustomEvent;
   ```

   c. Si vous utilisez la version générique, vous n'avez pas besoin de délégué personnalisé. Au lieu de cela, dans votre classe de publication, vous spécifiez votre type d'événement en tant que `EventHandler<CustomEventArgs>`, en insérant le nom de votre propre classe entre les crochets.

   ```
   public event EventHandler<CustomEventArgs> RaiseCustomEvent;
   ```

Exemple

L'exemple suivant est une illustration des étapes précédentes avec l'utilisation d'une classe EventArgs personnalisée et de EventHandler<TEventArgs> comme type d'événement.

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
```

```
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;

        public Subscriber(string id, Publisher pub)
        {
            _id = id;

            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }

    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}
```

Voir aussi

- Delegate
- Guide de programmation C#
- Événements
- Délégués

Comment déclencher des événements de classe de base dans des classes dérivées (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

L'exemple suivant montre la méthode standard employée pour déclarer des événements dans une classe de base pour qu'ils puissent être déclenchés à partir de classes dérivées. Ce modèle est largement utilisé dans les classes Windows Forms dans les bibliothèques de classes .NET.

Lorsque vous créez une classe qui peut être utilisée comme classe de base pour d'autres classes, tenez compte du fait que les événements constituent un type spécial de délégué qui ne peut être appelé qu'à partir de la classe qui les a déclarés. Les classes dérivées ne peuvent pas appeler directement les événements qui sont déclarés dans la classe de base. Même si vous pouvez avoir besoin d'un événement qui ne peut être déclenché que par la classe de base, la plupart du temps, vous devez permettre à la classe dérivée d'appeler les événements de la classe de base. Pour ce faire, vous pouvez créer une méthode d'appel protégée dans la classe de base qui encapsule l'événement. En appelant ou en substituant cette méthode d'appel, les classes dérivées peuvent appeler indirectement l'événement.

> **NOTE**
>
> Ne déclarez pas les événements virtuels dans une classe de base et substituez-les dans une classe dérivée. Le compilateur C# ne gère pas correctement ce type d'événement et il n'est pas possible de prévoir si un abonné de l'événement dérivé s'abonnera à l'événement de classe de base.

Exemple

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
```

```
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
```

```
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now {e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
```

```
        Received event. Shape area is now 49
        Drawing a rectangle
*/
```

Voir aussi

- Guide de programmation C#
- Événements
- Délégués
- Modificateurs d’accès
- Création de gestionnaires d'événements dans les Windows Forms

Comment implémenter des événements d'interface (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Une interface peut déclarer un événement. L'exemple suivant montre comment implémenter des événements d'interface dans une classe. En gros, les règles sont les mêmes que quand vous implémentez une propriété ou une méthode d'interface.

Pour implémenter des événements d'interface dans une classe

Déclarez l'événement dans votre classe, puis appelez-le aux emplacements appropriés.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Exemple

L'exemple suivant montre comment gérer la situation peu courante dans laquelle votre classe hérite de plusieurs interfaces et chaque interface a un événement ayant le même nom. Dans ce cas, vous devez fournir une implémentation d'interface explicite pour au moins l'un des événements. Quand vous écrivez une implémentation d'interface explicite pour un événement, vous devez également écrire les accesseurs d'événement `add` et `remove`. Normalement, ces éléments sont fournis par le compilateur, mais dans le cas en question le compilateur ne peut pas les fournir.

En fournissant vos propres accesseurs, vous pouvez spécifier si les deux événements sont représentés par le même événement dans votre classe ou par des événements différents. Par exemple, si les événements doivent être déclenchés à des moments différents en fonction des spécifications de l'interface, vous pouvez associer chaque événement à une implémentation distincte dans votre classe. Dans l'exemple suivant, les abonnés

déterminent l'événement `OnDraw` qu'il recevront en effectuant un cast de la référence de la forme en `IShape` ou `IDrawingObject`.

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
```

```
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

Voir aussi

- Guide de programmation C#
- Événements
- Délégués
- Implémentation d'interface explicite
- Comment déclencher des événements de la classe de base dans les classes dérivées

Comment implémenter des accesseurs d'événement personnalisés (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Un événement constitue un genre spécial de délégué multicast qui peut être appelé uniquement à partir de la classe dans laquelle il est déclaré. Le code client s'abonne à l'événement en fournissant une référence à une méthode qui doit être appelée quand l'événement est déclenché. Ces méthodes sont ajoutées à la liste d'invocation du délégué par le biais des accesseurs d'événement, qui ressemblent aux accesseurs de propriété, sauf que les accesseurs d'événement sont nommés `add` et `remove`. Dans la plupart des cas, vous n'avez pas à fournir d'accesseurs d'événement personnalisés. Quand aucun accesseur d'événement personnalisé n'est fourni dans votre code, le compilateur les ajoute automatiquement. Toutefois, vous devez parfois fournir un comportement personnalisé. L'un de ces cas est illustré dans la rubrique How to implement interface Events.

Exemple

L'exemple suivant indique comment implémenter des accesseurs d'événement add et remove. Même si vous pouvez remplacer le code à l'intérieur des accesseurs, nous vous recommandons de verrouiller l'événement avant d'ajouter ou de supprimer une nouvelle méthode de gestionnaire d'événements.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

Voir aussi

- Événements
- event

Génériques (guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Les génériques introduisent le concept de paramètres de type dans .NET, ce qui permet de concevoir des classes et des méthodes qui diffèrent la spécification d'un ou plusieurs types jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code client. Par exemple, en utilisant un paramètre de type générique `T`, vous pouvez écrire une seule classe qui peut être utilisée par un autre code client sans impliquer le coût ou le risque des casts ou des opérations de boxing du runtime, comme illustré ici :

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Les classes et les méthodes génériques combinent la réutilisabilité, la cohérence des types et l'efficacité d'une façon que leurs équivalents non génériques ne peuvent pas. Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux. L' System.Collections.Generic espace de noms contient plusieurs classes de collection génériques. Les collections non génériques, telles que, ne ArrayList sont pas recommandées et sont conservées à des fins de compatibilité. Pour plus d'informations, consultez Génériques en .NET.

Bien sûr, vous pouvez également créer des types et des méthodes génériques personnalisés pour fournir des solutions et des modèles de conception généralisés qui soient efficaces et de type sécurisé. L'exemple de code suivant montre une classe de liste liée générique simple, à des fins de démonstration. (Dans la plupart des cas, vous devez utiliser la List<T> classe fournie par .net au lieu de créer la vôtre.) Le paramètre `T` de type est utilisé dans plusieurs emplacements où un type concret est normalement utilisé pour indiquer le type de l'élément stocké dans la liste. Il est utilisé de la façon suivante :

- Comme le type d'un paramètre de méthode dans la méthode `AddHead`.
- Comme le type de retour de la propriété `Data` de la classe imbriquée `Node`.
- Comme le type de membre privé `data` de la classe imbriquée.

Notez que `T` est disponible pour la classe imbriquée `Node`. Quand `GenericList<T>` est instancié avec un type concret, par exemple comme un `GenericList<int>`, chaque occurrence de `T` est remplacée par `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

L'exemple de code suivant montre comment le code client utilise la classe générique `GenericList<T>` pour créer une liste d'entiers. En changeant l'argument de type, vous pouvez facilement modifier le code suivant pour créer des listes de chaînes ou tout autre type personnalisé :

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Vue d'ensemble des génériques

- Utilisez des types génériques pour optimiser la réutilisation de code, la sécurité des types et les performances.
- L'utilisation la plus courante des génériques est de créer des classes de collection.
- La bibliothèque de classes .NET contient plusieurs classes de collection génériques dans l' System.Collections.Generic espace de noms. Celles-ci doivent être utilisées chaque fois que c'est possible, à la place de classes comme ArrayList dans l'espace de noms System.Collections.
- Vous pouvez créer vos propres interfaces, classes, méthodes, événements et délégués génériques.
- Les classes génériques peuvent être contraintes pour permettre l'accès à des méthodes sur des types de données particuliers.
- Des informations sur les types qui sont utilisés dans un type de données générique peuvent être obtenues à l'exécution à l'aide de la réflexion.

Sections connexes

- Paramètres de type générique
- Contraintes sur les paramètres de type
- Classes génériques
- Interfaces génériques
- Méthodes génériques
- Délégués génériques
- Différences entre les modèles C++ et les génériques C#
- Génériques et réflexion
- Génériques dans le runtime

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#

- Types
- <typeparam>
- <typeparamref>
- Génériques en .NET

Paramètres de type générique (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Dans une définition de méthode ou de type générique, le paramètre de type représente un espace réservé pour un type spécifié par le client au moment de créer une instance du type générique. Une classe générique, telle que `GenericList<T>` répertoriée dans Introduction aux génériques, ne peut pas être utilisée en l'état car il ne s'agit pas vraiment d'un type, mais plutôt d'un modèle pour un type. Pour utiliser `GenericList<T>`, le code client doit déclarer et instancier un type construit en spécifiant un argument de type à l'intérieur de crochets pointus. L'argument de type pour cette classe particulière peut être tout type reconnu par le compilateur. Il est possible de créer un nombre quelconque d'instances de type construit, chacune avec un argument de type différent, comme suit :

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

Dans chacune de ces instances de `GenericList<T>`, chaque occurrence de `T` dans la classe est remplacée à l'exécution par l'argument de type. Par cette substitution, nous avons créé trois objets distincts de type sécurisé et efficaces à l'aide d'une seule définition de classe. Pour plus d'informations sur la façon dont cette substitution est exécutée par le CLR, consultez Génériques dans le runtime.

Recommandations concernant le nom des paramètres de type

- **Nommez** les paramètres de type générique avec des noms descriptifs, sauf si un nom composé d'une seule lettre est suffisamment évocateur et qu'un nom descriptif n'apporte rien de plus.

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Envisagez** l'utilisation de T comme nom de paramètre de type pour les types ayant un paramètre de type d'une seule lettre.

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Préfixez** les noms des paramètres de type descriptifs avec « T ».

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Pensez** à indiquer les contraintes placées sur un paramètre de type dans le nom de paramètre. Par exemple, un paramètre limité à `ISession` peut être appelé `TSession`.

La règle d'analyse du code CA1715 vérifie que les paramètres de type sont correctement nommés.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Différences entre les modèles C++ et les génériques C#

Contraintes sur les paramètres de type (Guide de programmation C#)

16/12/2020 • 19 minutes to read • Edit Online

Les contraintes informent le compilateur sur les fonctionnalités que doit avoir un argument de type. Sans contrainte, l'argument de type peut être n'importe quel type. Le compilateur peut seulement deviner les membres de System.Object, qui est la classe de base par excellence de tous les types .NET. Pour plus d'informations, consultez Pourquoi utiliser des contraintes. Si le code client utilise un type qui ne satisfait pas une contrainte, le compilateur émet une erreur. Les contraintes sont spécifiées à l'aide du mot clé contextuel `where`. Le tableau suivant répertorie les différents types de contraintes :

CONTRAINTE	DESCRIPTION
`where T : struct`	L'argument de type doit être un type valeurn'acceptant pas les valeurs NULL. Pour plus d'informations sur les types valeur Nullable, consultez types valeur Nullable. Étant donné que tous les types valeur ont un constructeur sans paramètre accessible, la `struct` contrainte implique la `new()` contrainte et ne peut pas être combinée avec la `new()` contrainte. Vous ne pouvez pas associer la `struct` contrainte à la `unmanaged` contrainte.
`where T : class`	L'argument de type doit être un type référence. Cette contrainte s'applique également à tous les types de classe, d'interface, de délégué ou de tableau. Dans un contexte Nullable en C# 8,0 ou une version ultérieure, `T` doit être un type de référence non Nullable.
`where T : class?`	L'argument de type doit être un type référence, Nullable ou non Nullable. Cette contrainte s'applique également à tous les types de classe, d'interface, de délégué ou de tableau.
`where T : notnull`	L'argument de type doit être un type non Nullable. L'argument peut être un type référence non Nullable en C# 8,0 ou version ultérieure, ou un type valeur qui n'autorise pas les valeurs NULL.
`where T : unmanaged`	L'argument de type doit être un type non managéqui n'accepte pas les valeurs NULL. La `unmanaged` contrainte implique la `struct` contrainte et ne peut pas être combinée avec `struct` les `new()` contraintes ou.
`where T : new()`	L'argument de type doit avoir un constructeur sans paramètre public. Quand vous utilisez la contrainte `new()` avec d'autres contraintes, elle doit être spécifiée en dernier. La `new()` contrainte ne peut pas être combinée avec les `struct` `unmanaged` contraintes et.
`where T :` *<base class name>*	L'argument de type doit être la classe de base spécifiée ou en dériver. Dans un contexte Nullable en C# 8,0 et versions ultérieures, `T` doit être un type de référence non Nullable dérivé de la classe de base spécifiée.

CONTRAINTE	DESCRIPTION
`where T :` *<base class name>?*	L'argument de type doit être la classe de base spécifiée ou en dériver. Dans un contexte Nullable en C# 8,0 et versions ultérieures, `T` peut être un type Nullable ou non Nullable dérivé de la classe de base spécifiée.
`where T :` *<interface name>*	L'argument de type doit être ou implémenter l'interface spécifiée. Plusieurs contraintes d'interface peuvent être spécifiées. L'interface qui impose les contraintes peut également être générique. Dans un contexte Nullable en C# 8,0 et versions ultérieures, `T` doit être un type non Nullable qui implémente l'interface spécifiée.
`where T :` *<interface name>?*	L'argument de type doit être ou implémenter l'interface spécifiée. Plusieurs contraintes d'interface peuvent être spécifiées. L'interface qui impose les contraintes peut également être générique. Dans un contexte Nullable en C# 8,0, `T` peut être un type de référence Nullable, un type de référence non Nullable ou un type valeur. `T` n'est pas un type valeur Nullable.
`where T : U`	L'argument de type fourni pour `T` doit être ou dériver de l'argument fourni pour `U` . Dans un contexte Nullable, si `U` est un type référence qui n'accepte pas les valeurs NULL, `T` doit être un type référence non Nullable. Si `U` est un type de référence Nullable, `T` peut avoir la valeur null ou n'accepte pas les valeurs NULL.

Pourquoi utiliser des contraintes

Les contraintes spécifient les capacités et les attentes d'un paramètre de type. La déclaration de ces contraintes signifie que vous pouvez utiliser les opérations et les appels de méthode du type de contrainte. Si votre classe ou méthode générique utilise une opération sur les membres génériques au-delà de l'assignation simple ou l'appel de toute méthode non prise en charge par System.Object , vous devez appliquer des contraintes au paramètre de type. Par exemple, la contrainte de classe de base indique au compilateur que seuls les objets de ce type ou dérivés de ce type seront utilisés comme arguments de type. Une fois que le compilateur a cette garantie, il peut autoriser les méthodes de ce type à être appelées dans la classe générique. L'exemple de code suivant illustre la fonctionnalité que vous pouvez ajouter à la classe `GenericList<T>` (dans Introduction aux génériques) en appliquant une contrainte de classe de base.

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

La contrainte permet à la classe générique d'utiliser la propriété `Employee.Name`. La contrainte spécifie que tous les éléments de type `T` sont soit un objet `Employee`, soit un objet qui hérite de `Employee`, et rien d'autre.

Plusieurs contraintes peuvent être appliquées au même paramètre de type, et les contraintes elles-mêmes peuvent être des types génériques, comme suit :

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

En appliquant la contrainte `where T : class`, évitez d'utiliser les opérateurs `==` et `!=` sur le paramètre de type, car ces opérateurs testent uniquement l'identité des références, et non l'égalité des valeurs. Ce comportement se produit même si ces opérateurs sont surchargés dans un type qui est utilisé comme argument. Le code suivant illustre ce point ; la sortie a la valeur false même si la classe String surcharge l'opérateur `==`.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

Le compilateur sait uniquement que `T` est un type référence au moment de la compilation et doit utiliser les opérateurs par défaut qui sont valides pour tous les types référence. Si vous devez tester l'égalité des valeurs, il est recommandé d'appliquer également la contrainte `where T : IEquatable<T>` ou `where T : IComparable<T>` et d'implémenter l'interface dans toute classe qui sera utilisée pour construire la classe générique.

Utilisation de contraintes dans plusieurs paramètres

Vous pouvez appliquer des contraintes à plusieurs paramètres et plusieurs contraintes à un seul paramètre, comme indiqué dans l'exemple suivant :

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Paramètres de type unbounded

Les paramètres de type qui n'ont aucune contrainte, tels que T dans la classe publique `SampleClass<T>{}`, sont appelés paramètres de type unbounded. Les paramètres de type unbounded obéissent aux règles suivantes :

- Les `!=` `==` opérateurs et ne peuvent pas être utilisés, car il n'y a aucune garantie que l'argument de type concret prendra en charge ces opérateurs.
- Ils peuvent être convertis vers et depuis `System.Object` ou être explicitement convertis vers tout type d'interface.
- Vous pouvez les comparer à null. Si un paramètre unbounded est comparé à `null`, la comparaison retourne toujours la valeur false si l'argument de type est un type valeur.

Paramètres de type en tant que contraintes

L'utilisation d'un paramètre de type générique comme contrainte est utile quand une fonction membre dotée de son propre paramètre de type doit contraindre ce paramètre au paramètre du type conteneur, comme le montre

l'exemple suivant :

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

Dans l'exemple précédent, `T` est une contrainte de type dans le contexte de la méthode `Add` et un paramètre de type unbounded dans le contexte de la classe `List`.

Les paramètres de type peuvent également être utilisés comme contraintes dans les définitions de classes génériques. Le paramètre de type doit être déclaré entre crochets pointus, ainsi que tous les autres paramètres de type :

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

L'utilité des paramètres de type en tant que contraintes avec les classes génériques est limitée, car le compilateur ne peut rien deviner à propos du paramètre de type en dehors du fait qu'il dérive de `System.Object`. Utilisez des paramètres de type en tant que contraintes sur les classes génériques dans les scénarios dans lesquels vous souhaitez mettre en application une relation d'héritage entre deux paramètres de type.

Contrainte NotNull

À compter de C# 8,0 dans un contexte Nullable, vous pouvez utiliser la `notnull` contrainte pour spécifier que l'argument de type doit être un type valeur n'acceptant pas les valeurs null ou un type référence non Nullable. La `notnull` contrainte ne peut être utilisée que dans un `nullable enable` contexte. Le compilateur génère un avertissement si vous ajoutez la `notnull` contrainte dans un contexte oublie Nullable.

Contrairement à d'autres contraintes, lorsqu'un argument de type viole la `notnull` contrainte, le compilateur génère un avertissement lorsque ce code est compilé dans un `nullable enable` contexte. Si le code est compilé dans un contexte oublie Nullable, le compilateur ne génère pas d'avertissements ni d'erreurs.

À compter de C# 8,0 dans un contexte Nullable, la `class` contrainte spécifie que l'argument de type doit être un type de référence n'acceptant pas les valeurs NULL. Dans un contexte Nullable, lorsqu'un paramètre de type est un type référence Nullable, le compilateur génère un avertissement.

Contrainte non managée

À compter de C# 7,3, vous pouvez utiliser la `unmanaged` contrainte pour spécifier que le paramètre de type doit être un type non managéqui n'accepte pas les valeurs NULL. La contrainte `unmanaged` vous permet d'écrire des routines réutilisables à appliquer aux types qui peuvent être manipulés comme blocs de mémoire, comme illustré dans l'exemple suivant :

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

La méthode précédente doit être compilée dans un contexte `unsafe`, car elle utilise l'opérateur `sizeof` sur un

type qui n'est pas connu pour être un type intégré. Sans la contrainte `unmanaged`, l'opérateur `sizeof` n'est pas disponible.

La `unmanaged` contrainte implique la `struct` contrainte et ne peut pas être associée à celle-ci. Étant donné que la `struct` contrainte implique la `new()` contrainte, la contrainte `unmanaged` ne peut pas être combinée à la contrainte `new()` .

Contraintes de délégué

À partir de C# 7.3, vous pouvez aussi utiliser System.Delegate ou System.MulticastDelegate comme contrainte de classe de base. Le CLR a toujours autorisé cette contrainte, contrairement au langage C#. La contrainte `System.Delegate` vous permet d'écrire du code qui fonctionne avec les délégués en mode type sécurisé. Le code suivant définit une méthode d'extension qui combine deux délégués à condition qu'ils soient du même type :

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

Vous pouvez utiliser la méthode ci-dessus pour combiner des délégués qui sont du même type :

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Si vous supprimez les commentaires de la dernière ligne, il ne sera pas compilé. `first` Et `test` sont des types délégués, mais il s'agit de types délégués différents.

Contraintes d'enum

À compter de C# 7.3, vous pouvez également spécifier le type System.Enum comme contrainte de classe de base. Le CLR a toujours autorisé cette contrainte, contrairement au langage C#. Les génériques utilisant `System.Enum` fournissent une programmation de type sécurisé aux résultats de cache issus de l'utilisation de méthodes statiques dans `System.Enum`. L'exemple suivant recherche toutes les valeurs valides d'un type enum, puis génère un dictionnaire qui mappe ces valeurs à sa représentation sous forme de chaîne.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

`Enum.GetValues` et `Enum.GetName` utilisent la réflexion, ce qui a des répercussions sur les performances. Vous pouvez appeler `EnumNamedValues` pour générer une collection qui est mise en cache et réutilisée au lieu de répéter les appels qui requièrent la réflexion.

Vous pouvez l'utiliser comme montré dans l'exemple suivant pour créer un enum et générer un dictionnaire de ses valeurs et de ses noms :

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Classes génériques
- nouvelle contrainte

Classes génériques (guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Les classes génériques encapsulent des opérations qui ne sont pas spécifiques à un type de données particulier. Les classes génériques sont le plus souvent utilisées avec les collections telles que les listes liées, les tables de hachage, les piles, les files d'attente, les arborescences, etc. Les opérations telles que l'ajout et la suppression d'éléments dans la collection sont exécutées fondamentalement de la même manière quel que soit le type des données stockées.

Pour la plupart des scénarios qui nécessitent des classes de collections, l'approche recommandée consiste à utiliser celles fournies dans la bibliothèque de classes .NET. Pour plus d'informations sur l'utilisation de ces classes, consultez Collections génériques dans .NET.

En général, vous créez des classes génériques en commençant par une classe concrète existante et en changeant des types en paramètres de type individuellement jusqu'à ce que vous atteigniez l'équilibre optimal au niveau de la généralisation et de la facilité d'utilisation. Lors de la création de vos propres classes génériques, les aspects importants à prendre en considération sont notamment les suivants :

- Quels types généraliser en paramètres de type.

  En règle générale, plus le nombre de types que vous pouvez paramétrer est important, plus votre code est souple et réutilisable. Toutefois, une généralisation excessive peut produire un code qui est difficile à lire ou à comprendre par d'autres développeurs.

- Quelles contraintes, le cas échéant, appliquer aux paramètres de type (consultez Contraintes sur les paramètres de type).

  Il est pertinent d'appliquer le maximum de contraintes possible vous permettant néanmoins de gérer les types que vous devez gérer. Par exemple, si vous savez que votre classe générique est destinée uniquement à être utilisée avec les types référence, appliquez la contrainte de classe. Cela empêchera l'utilisation involontaire de votre classe avec les types valeur et vous permettra d'utiliser l'opérateur `as` sur `T` et de rechercher les valeurs nulles.

- S'il convient de prendre en compte le comportement générique dans les sous-classes et les classes de base.

  Les classes génériques pouvant servir de classes de base, les considérations de conception qui s'appliquent ici sont les mêmes que pour les classes non génériques. Consultez les règles relatives à l'héritage à partir des classes de base génériques plus loin dans cette rubrique.

- S'il convient d'implémenter une ou plusieurs interfaces génériques.

  Par exemple, si vous concevez une classe qui sera utilisée pour créer des éléments dans une collection basée sur les génériques, vous pouvez avoir besoin d'implémenter une interface comme IComparable<T>, où `T` est le type de votre classe.

Pour obtenir un exemple d'une classe générique simple, consultez Introduction aux génériques.

Les règles pour les paramètres de type et les contraintes ont plusieurs conséquences sur le comportement de classe générique, surtout en ce qui concerne l'héritage et l'accessibilité des membres. Avant de continuer, vous devez comprendre quelques termes. Pour une classe générique `Node<T>,` , le code client peut référencer la classe soit en spécifiant un argument de type, pour créer un type construit fermé (`Node<int>`), soit en laissant le paramètre de type non spécifié, par exemple quand vous spécifiez une classe de base générique, pour créer un type construit ouvert (`Node<T>`). Les classes génériques peuvent hériter de classes de base concrètes, construites

fermées ou construites ouvertes :

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Les classes non génériques, également appelées concrètes, peuvent hériter des classes de base construites fermées, mais pas des classes construites ouvertes ni des paramètres de type parce que le code client ne peut en aucune façon fournir au moment de l'exécution l'argument de type requis pour instancier la classe de base.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Les classes génériques qui héritent de types construits ouverts doivent fournir des arguments de type pour tout paramètre de type de classe de base qui n'est pas partagé par la classe qui hérite, comme cela est indiqué dans le code suivant :

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Les classes génériques qui héritent des types construits ouverts doivent spécifier des contraintes qui sont un sur-ensemble, ou qui impliquent, des contraintes sur le type de base :

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Les types génériques peuvent utiliser plusieurs paramètres de type et contraintes, comme suit :

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Les types construits ouverts et les types construits fermés peuvent être utilisés comme paramètres de méthode :

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Si une classe générique implémente une interface, toutes les instances de cette classe peuvent être castées à cette interface.

Les classes génériques sont indifférentes. En d'autres termes, si un paramètre d'entrée spécifie un `List<BaseClass>`, vous obtiendrez une erreur de compilation si vous essayez de fournir un `List<DerivedClass>`.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Enregistrement de l'état des énumérateurs
- An Inheritance Puzzle, Part One

Interfaces génériques (guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Il est souvent utile de définir des interfaces pour les classes de collections génériques, ou pour les classes génériques qui représentent des éléments dans la collection. Pour les classes génériques, il est préférable d'utiliser des interfaces génériques, comme IComparable<T> à la place d'IComparable, pour éviter les opérations de boxing et d'unboxing sur les types valeur. La bibliothèque de classes .NET définit plusieurs interfaces génériques à utiliser avec les classes de collection dans l' System.Collections.Generic espace de noms.

Quand une interface est spécifiée comme une contrainte sur un paramètre de type, seuls les types qui implémentent l'interface peuvent être utilisés. L'exemple de code suivant illustre une classe `SortedList<T>` qui dérive de la classe `GenericList<T>`. Pour plus d'informations, consultez Introduction aux génériques. `SortedList<T>` ajoute la contrainte `where T : IComparable<T>`. Ceci permet à la méthode `BubbleSort` dans `SortedList<T>` d'utiliser la méthode générique CompareTo sur les éléments de liste. Dans cet exemple, les éléments de liste sont une classe simple, `Person`, qui implémente `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }
```

```
    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
```

```
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
```

```
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Plusieurs interfaces peuvent être spécifiées en tant que contraintes sur un seul type, comme suit :

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Une interface peut définir plusieurs paramètres de type, comme suit :

```
interface IDictionary<K, V>
{
}
```

Les règles d'héritage qui s'appliquent aux classes s'appliquent également aux interfaces :

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

Les interfaces génériques peuvent hériter d'interfaces non génériques si l'interface générique est de type contravariant, ce qui signifie qu'elle utilise uniquement son paramètre de type comme valeur de retour. Dans la bibliothèque de classes .NET, IEnumerable<T> hérite de IEnumerable , car IEnumerable<T> utilise uniquement `T` dans la valeur de retour de GetEnumerator et dans l' Current accesseur Get de propriété.

Les classes concrètes peuvent implémenter des interfaces construites fermées, comme suit :

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Les classes génériques peuvent implémenter des interfaces génériques ou des interfaces construites fermées tant que la liste de paramètres de classe fournit tous les arguments requis par l'interface, comme suit :

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

Les règles qui déterminent la surcharge des méthodes sont les mêmes pour les méthodes dans les classes

génériques, les structs génériques ou les interfaces génériques. Pour plus d'informations, consultez Méthodes génériques.

Voir aussi

- Guide de programmation C#
- Introduction aux génériques
- interface
- Génériques

Méthodes génériques (guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Une méthode générique est une méthode qui est déclarée avec des paramètres de type, comme suit :

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

L'exemple de code suivant montre une manière d'appeler la méthode, avec `int` comme argument de type :

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

Vous pouvez également omettre l'argument de type et le compilateur le déduira. L'appel suivant à `Swap` est équivalent à l'appel précédent :

```
Swap(ref a, ref b);
```

Les mêmes règles d'inférence de type s'appliquent aux méthodes statiques et aux méthodes d'instance. Le compilateur peut déduire les paramètres de type d'après les arguments de méthode que vous passez. Il ne peut pas déduire les paramètres de type uniquement à partir d'une valeur de contrainte ou de retour. Par conséquent, l'inférence de type ne fonctionne pas avec les méthodes qui n'ont pas de paramètres. L'inférence de type se produit au moment de la compilation avant que le compilateur n'essaie de résoudre toute signature de méthode surchargée. Le compilateur applique la logique d'inférence de type à toutes les méthodes génériques qui partagent le même nom. À l'étape de résolution de la surcharge, le compilateur inclut uniquement les méthodes génériques sur lesquelles l'inférence de type a réussi.

Dans une classe générique, les méthodes non génériques peuvent accéder aux paramètres de type de niveau classe, comme suit :

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Si vous définissez une méthode générique qui accepte les mêmes paramètres de type que la classe conteneur, le compilateur génère l'avertissement CS0693, car l'argument fourni à l'intérieur de la portée de la méthode pour le `T` interne masque l'argument fourni pour le `T` externe. Si vous avez éventuellement besoin d'appeler une méthode de classe générique avec des arguments de type différents de ceux qui ont été fournis quand la classe a été instanciée, envisagez de fournir un autre identificateur pour le paramètre de type de la méthode, comme

indiqué dans `GenericList2<T>` dans l'exemple suivant.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

Utilisez des contraintes pour permettre des opérations plus spécialisées sur les paramètres de type dans les méthodes. Cette version de `Swap<T>`, maintenant appelée `SwapIfGreater<T>`, peut être utilisée avec des arguments de type qui implémentent IComparable<T> uniquement.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Les méthodes génériques peuvent être surchargées sur plusieurs paramètres de type. Par exemple, les méthodes suivantes peuvent toutes être dans la même classe :

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Méthodes

Génériques et tableaux (guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

En C# 2.0 et versions ultérieures, les tableaux unidimensionnels qui ont une limite inférieure de zéro implémentent automatiquement IList<T>. Cela vous permet de créer des méthodes génériques qui peuvent utiliser le même code pour itérer au sein de tableaux et d'autres types de collections. Cette technique est essentiellement utile pour lire les données dans des collections. L'interface IList<T> ne peut pas être utilisée pour ajouter ni supprimer des éléments dans un tableau. Une exception est levée si vous essayez d'appeler une méthode IList<T> telle que RemoveAt sur un tableau dans ce contexte.

L'exemple de code suivant montre comment une méthode générique seule qui prend un paramètre d'entrée IList<T> peut itérer à la fois au sein d'une liste et d'un tableau (en l'occurrence un tableau d'entiers).

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Tableaux
- Génériques

Délégués génériques (guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Un délégué peut définir ses propres paramètres de type. Le code qui référence le délégué générique peut spécifier l'argument de type pour créer un type construit fermé, comme lors de l'instanciation d'une classe générique ou d'un appel d'une méthode générique, ainsi que l'illustre l'exemple ci-dessous :

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

Le langage C# version 2.0 propose une nouvelle fonctionnalité appelée conversion de groupe de méthodes, qui s'applique aussi bien aux types concrets qu'aux types délégués génériques et vous permet d'écrire la ligne précédente avec la syntaxe simplifiée suivante :

```
Del<int> m2 = Notify;
```

Les délégués définis dans une classe générique peuvent utiliser les paramètres de type de classe générique, à l'instar des méthodes de classe.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

Le code qui référence le délégué doit spécifier l'argument de type de la classe conteneur, comme suit :

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Les délégués génériques sont particulièrement utiles pour définir des événements basés sur le modèle de design type parce que l'argument de l'expéditeur peut être fortement typé et il n'est plus nécessaire d'en effectuer un cast vers et depuis Object.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Méthodes génériques
- Classes génériques
- Interfaces génériques
- Délégués
- Génériques

Différences entre les modèles C++ et les génériques C# (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Les génériques C# et les modèles C++ sont des fonctionnalités linguistiques qui prennent en charge les types paramétrables. Toutefois, il existe de nombreuses différences entre les deux. Au niveau de la syntaxe, les génériques C# constituent une approche plus simple des types paramétrables, sans la complexité des modèles C++. De plus, C# n'essaie pas de fournir toutes les fonctionnalités fournies par les modèles C++. Au niveau de l'implémentation, la principale différence est que les substitutions de type générique C# sont effectuées au moment de l'exécution. Ainsi, les informations de type générique sont conservées pour les objets instanciés. Pour plus d'informations, consultez Génériques dans le runtime.

Voici les principales différences entre les modèles C++ et les génériques C# :

- Les génériques C# n'offrent pas le même niveau de flexibilité que les modèles C++. Par exemple, il n'est pas possible d'appeler des opérateurs arithmétiques dans une classe générique C#, bien qu'il soit possible d'appeler des opérateurs définis par l'utilisateur.
- C# n'autorise pas les paramètres de modèle sans type, tels que `template C<int i> {}`.
- C# ne prend pas en charge la spécialisation explicite (une implémentation personnalisée d'un modèle pour un type spécifique).
- C# ne prend pas en charge la spécialisation partielle (une implémentation personnalisée pour un sous-ensemble des arguments de type).
- C# n'autorise pas l'utilisation du paramètre de type comme classe de base pour le type générique.
- C# n'autorise pas les paramètres de type à avoir des types par défaut.
- En C#, un paramètre de type générique ne peut pas être lui-même un générique, bien que les types construits puissent être utilisés comme des génériques. C++ autorise les paramètres de modèle.
- C++ autorise le code qui peut ne pas être valide pour tous les paramètres de type dans le modèle, qui est ensuite vérifié pour identifier le type spécifique utilisé comme paramètre de type. C# exige que le code dans une classe soit écrit de telle sorte qu'il fonctionne avec tout type qui satisfait aux contraintes. Par exemple, en C++ il est possible d'écrire une fonction qui utilise les opérateurs arithmétiques `+` et `-` sur des objets du paramètre de type, ce qui générera une erreur au moment de l'instanciation du modèle avec un type qui ne prend pas en charge ces opérateurs. Ceci n'est pas autorisé par C#. Les seules constructions de langage autorisées sont celles qui peuvent être déduites des contraintes.

Voir aussi

- Guide de programmation C#
- Introduction aux génériques
- Modèles

Génériques dans le runtime (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Quand une méthode ou un type générique est compilé dans le langage intermédiaire Microsoft (MSIL), il ou elle contient des métadonnées qui l'identifient comme ayant des paramètres de type. La façon dont le langage MSIL d'un type générique est utilisé diffère selon que le paramètre de type fourni est un type valeur ou référence.

Quand un type générique est construit en premier avec un type valeur comme paramètre, le runtime crée un type générique spécialisé avec le paramètre fourni ou les paramètres substitués aux emplacements appropriés dans le langage MSIL. Des types génériques spécialisés sont créés une fois pour chaque type valeur unique qui est utilisé comme paramètre.

Par exemple, supposons que votre code de programme ait déclaré une pile construite d'entiers :

```
Stack<int> stack;
```

À ce stade, le runtime génère une version spécialisée de la classe Stack<T> dont l'entier a été substitué correctement à son paramètre. Maintenant, chaque fois que le code de programme utilise une pile d'entiers, le runtime réutilise la classe Stack<T> spécialisée générée. Dans l'exemple suivant, deux instances d'une pile d'entiers sont créées, et elles partagent une instance unique du code `Stack<int>` :

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

Toutefois, supposons qu'une autre classe Stack<T> avec un type valeur différent tel qu'un type `long` ou une structure définie par l'utilisateur comme son paramètre est créée à un autre endroit dans votre code. Le runtime génère alors une autre version du type générique et substitue un type `long` aux emplacements appropriés dans le langage MSIL. Les conversions ne sont plus nécessaires parce que chaque classe générique spécialisée contient le type valeur en mode natif.

Les génériques fonctionnent un peu différemment pour les types référence. La première fois qu'un type générique est construit avec un type référence quelconque, le runtime crée un type générique spécialisé avec les références d'objet substituées aux paramètres dans le langage MSIL. Ensuite, chaque fois qu'un type construit est instancié avec un type référence comme son paramètre, indépendamment de son type, le runtime réutilise la version spécialisée créée précédemment pour le type générique. Ceci est possible parce que toutes les références ont la même taille.

Par exemple, supposons que vous ayez deux types référence, une classe `Customer` et une classe `Order`, et supposons également que vous ayez créé une pile de types `Customer` :

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

À ce stade, le runtime génère une version spécialisée de la classe Stack<T> qui stocke des références d'objet qui seront remplies ultérieurement au lieu de stocker des données. Supposons que la ligne de code suivante crée une

pile d'un autre type référence, qui est nommé `Order` :

```
Stack<Order> orders = new Stack<Order>();
```

Contrairement aux types valeur, une autre version spécialisée de la classe Stack<T> n'est pas créée pour le type `Order`. À la place, une instance de la version spécialisée de la classe Stack<T> est créée et la variable `orders` est définie pour la référencer. Supposons que vous ayez rencontré ensuite une ligne de code pour créer une pile d'un type `Customer` :

```
customers = new Stack<Customer>();
```

Comme avec l'utilisation précédente de la classe Stack<T> créée à l'aide du type `Order`, une autre instance de la classe Stack<T> spécialisée est créée. Les pointeurs contenus ici sont configurés pour référencer une zone de mémoire de la taille d'un type `Customer`. Dans la mesure où le nombre de types référence peut varier de manière importante d'un programme à l'autre, l'implémentation C# de génériques réduit sensiblement le volume du code en ramenant à une unité le nombre de classes spécialisées créées par le compilateur pour les classes génériques de types référence.

De plus, quand une classe C# générique est instanciée à l'aide d'un paramètre de type valeur ou référence, la réflexion peut l'interroger au moment de l'exécution, et son type réel ainsi que son paramètre de type peuvent être vérifiés.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Génériques

Génériques et réflexion (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

Étant donné que le common language runtime (CLR) a accès aux informations concernant les types génériques au moment de l'exécution, vous pouvez utiliser la réflexion pour obtenir des informations sur les types génériques de la même manière que pour les types non génériques. Pour plus d'informations, consultez Génériques dans le runtime.

Dans .NET Framework 2,0, plusieurs nouveaux membres ont été ajoutés à la Type classe pour activer les informations d'exécution pour les types génériques. Pour plus d'informations sur l'utilisation de ces méthodes et de ces propriétés, consultez la documentation relative à ces classes. L'espace de noms System.Reflection.Emit contient également de nouveaux membres qui prennent en charge les génériques. Consultez Comment : définir un type générique avec l'émission de réflexion.

Pour obtenir la liste des conditions indifférentes pour les termes utilisés dans la réflexion générique, consultez les notes sur la propriété IsGenericType.

NOM DE MEMBRE SYSTEM.TYPE	DESCRIPTION
IsGenericType	Retourne la valeur true si un type est générique.
GetGenericArguments	Retourne un tableau d'objets `Type` qui représentent les arguments de type fournis pour un type construit, ou les paramètres de type d'une définition de type générique.
GetGenericTypeDefinition	Retourne la définition de type générique sous-jacente pour le type construit actuel.
GetGenericParameterConstraints	Retourne un tableau d'objets `Type` qui représentent les contraintes qui s'exercent sur le paramètre de type générique actuel.
ContainsGenericParameters	Retourne la valeur true si le type ou l'un de ses types ou méthodes englobants contient des paramètres de type pour lesquels des types spécifiques n'ont pas été fournis.
GenericParameterAttributes	Obtient une combinaison d'indicateurs `GenericParameterAttributes` décrivant les contraintes spéciales du paramètre de type générique actuel.
GenericParameterPosition	Pour un objet `Type` qui représente un paramètre de type, obtient la position du paramètre de type dans la liste de paramètres de type de la définition de type générique ou de la définition de méthode générique qui a déclaré le paramètre de type.
IsGenericParameter	Obtient une valeur indiquant si le `Type` actuel représente un paramètre de type d'un type générique ou d'une définition de méthode.

NOM DE MEMBRE SYSTEM.TYPE	DESCRIPTION
IsGenericTypeDefinition	Obtient une valeur qui indique si le Type actuel représente une définition de type générique, à partir de laquelle d'autres types génériques peuvent être construits. Retourne la valeur true si le type représente la définition d'un type générique.
DeclaringMethod	Retourne la méthode générique qui a défini le paramètre de type générique actuel, ou Null si le paramètre de type n'a pas été défini par une méthode générique.
MakeGenericType	Substitue les éléments d'un tableau de types aux paramètres de type de la définition du type générique actuel et retourne un objet Type qui représente le type construit résultant.

De plus, les membres de la classe MethodInfo permettent la génération d'informations d'exécution pour les méthodes génériques. Pour obtenir la liste des conditions indifférentes des termes utilisés pour réfléchir les méthodes génériques, consultez les notes relatives à la propriété IsGenericMethod.

NOM DE MEMBRE SYSTEM.REFLECTION.MEMBERINFO	DESCRIPTION
IsGenericMethod	Retourne la valeur true si une méthode est générique.
GetGenericArguments	Retourne un tableau d'objets de type qui représentent les arguments de type d'une méthode générique construite, ou les paramètres de type d'une définition de méthode générique.
GetGenericMethodDefinition	Retourne la définition de méthode générique sous-jacente pour la méthode construite actuelle.
ContainsGenericParameters	Retourne la valeur true si la méthode ou l'un de ses types englobants contient des paramètres de type pour lesquels des types spécifiques n'ont pas été fournis.
IsGenericMethodDefinition	Retourne la valeur true si le MethodInfo actuel représente la définition d'une méthode générique.
MakeGenericMethod	Substitue les éléments d'un tableau de types aux paramètres de type de la définition de méthode générique actuelle et retourne un objet MethodInfo représentant la méthode construite résultante.

Voir aussi

- Guide de programmation C#
- Génériques
- Réflexion et types génériques
- Génériques

Génériques et attributs (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Les attributs peuvent être appliqués aux types génériques de la même manière qu'aux types non génériques. Pour plus d'informations sur l'application des attributs, consultez Attributs.

Les attributs personnalisés sont uniquement autorisés à référencer des types génériques ouverts, qui sont des types génériques pour lesquels aucun argument de type n'est fourni, et des types génériques construits fermés, qui fournissent des arguments pour tous les paramètres de type.

Les exemples suivants utilisent cet attribut personnalisé :

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

Un attribut peut référencer un type générique ouvert :

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Spécifiez plusieurs paramètres de type à l'aide du nombre approprié de virgules. Dans cet exemple, `GenericClass2` a deux paramètres de type :

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Un attribut peut référencer un type générique construit fermé :

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Un attribut qui référence un paramètre de type générique entraîne une erreur de compilation :

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
class ClassD<T> { }
```

Un type générique ne peut pas hériter d'Attribute :

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

Pour obtenir des informations sur un type générique ou un paramètre de type au moment de l'exécution, vous pouvez utiliser les méthodes de System.Reflection. Pour plus d'informations, consultez Génériques et réflexion

Voir aussi

- Guide de programmation C#
- Génériques
- Attributs

Espaces de noms (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Les espaces de noms sont largement utilisés de deux façons dans la programmation avec C#. Tout d'abord, .NET utilise des espaces de noms pour organiser ses nombreuses classes, comme suit :

```
System.Console.WriteLine("Hello World!");
```

System est un espace de noms et Console est une classe de cet espace de noms. Vous pouvez utiliser le mot clé `using`. Ainsi, le nom complet n'est pas obligatoire, comme dans l'exemple suivant :

```
using System;
```

```
Console.WriteLine("Hello World!");
```

Pour plus d'informations, consultez Directive using.

Deuxièmement, la déclaration de vos propres espaces de noms peut vous aider à contrôler l'étendue des noms de classes et de méthodes dans les projets de programmation plus vastes. Utilisez le mot clé namespace pour déclarer un espace de noms, comme dans l'exemple suivant :

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

Le nom de l'espace de noms doit être un nom d'identificateur C# valide.

Vue d'ensemble des espaces de noms

Les espaces de noms ont les propriétés suivantes :

- Ils organisent les projets de code de taille importante.
- Ils sont délimités à l'aide de l'opérateur `.`.
- La directive `using` évite de devoir spécifier le nom de l'espace de noms pour chaque classe.
- L'espace de noms `global` est l'espace de noms « racine » : `global::System` fait toujours référence à l'espace de noms System .NET.

spécification du langage C#

Pour plus d'informations, voir la section Espace de noms de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Utilisation d'espaces de noms
- Comment utiliser l'espace de noms My
- Noms d'identificateurs
- using, directive
- ::, Opérateur

Utilisation d'espaces de noms (Guide de programmation C#)

02/11/2020 • 7 minutes to read • Edit Online

Les espaces de noms sont largement utilisés dans les programmes C#, et ce de deux façons différentes. Tout d'abord, les classes .NET utilisent des espaces de noms pour organiser ses nombreuses classes. En deuxième lieu, la déclaration de vos propres espaces de noms peut faciliter le contrôle de la portée des noms de classes et de méthodes dans les projets de programmation de plus grande envergure.

Accès aux espaces de noms

La plupart des applications C# commencent par une section de directives `using`. Cette section répertorie les espaces de noms que l'application est appelée à utiliser fréquemment et évite au programmeur de spécifier un nom qualifié complet chaque fois qu'une méthode qu'elle contient est utilisée.

Par exemple, en incluant la ligne :

```
using System;
```

Au début d'un programme, le programmeur peut utiliser le code :

```
Console.WriteLine("Hello, World!");
```

À la place de :

```
System.Console.WriteLine("Hello, World!");
```

Alias d'espaces de noms

Vous pouvez également utiliser la `using` directive pour créer un alias pour un espace de noms. Utilisez le qualificateur d'alias d'espace de noms `::` pour accéder aux membres de l'espace de noms avec alias. L'exemple suivant montre comment créer et utiliser un alias d'espace de noms :

```
using generics = System.Collections.Generic;

namespace AliasExample
{
    class TestClass
    {
        static void Main()
        {
            generics::Dictionary<string, int> dict = new generics::Dictionary<string, int>()
            {
                ["A"] = 1,
                ["B"] = 2,
                ["C"] = 3
            };

            foreach (string name in dict.Keys)
            {
                System.Console.WriteLine($"{name} {dict[name]}");
            }
            // Output:
            // A 1
            // B 2
            // C 3
        }
    }
}
```

Utilisation d'espaces de noms pour contrôler la portée

Le mot clé `namespace` est utilisé pour déclarer une portée. La possibilité de créer des portées au sein de votre projet facilite l'organisation du code et vous permet de créer des types globalement uniques. Dans l'exemple suivant, une classe intitulée `SampleClass` est définie dans deux espaces de noms, l'un étant imbriqué à l'intérieur de l'autre. Le `.` jeton est utilisé pour distinguer la méthode qui est appelée.

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                  "SampleMethod inside NestedNamespace");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Displays "SampleMethod inside SampleNamespace."
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
            outer2.SampleMethod();

            // Displays "SampleMethod inside NestedNamespace."
            NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
            inner.SampleMethod();
        }
    }
}
```

Noms qualifiés complets

Les espaces de noms et les types portent des titres uniques décrits par des noms qualifiés complets qui indiquent une hiérarchie logique. Par exemple, l'instruction `A.B` implique que `A` est le nom de l'espace de noms ou du type et que `B` est imbriqué en son sein.

Dans l'exemple suivant, il y a des classes et des espaces de noms imbriqués. Le nom qualifié complet est indiqué sous forme de commentaire à la suite de chaque entité.

```
namespace N1     // N1
{
    class C1      // N1.C1
    {
        class C2   // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2   // N1.N2.C2
        {
        }
    }
}
```

Dans le segment de code précédent :

- L'espace de noms `N1` est un membre de l'espace de noms global. Son nom qualifié complet est `N1`.
- L'espace de noms `N2` est un membre de `N1`. Son nom qualifié complet est `N1.N2`.
- La classe `C1` est un membre de `N1`. Son nom qualifié complet est `N1.C1`.
- Le nom de classe `C2` est utilisé deux fois dans ce code. En revanche, les noms qualifiés complets sont uniques. La première instance de `C2` est déclarée à l'intérieur de `C1` ; par conséquent, son nom complet est : `N1.C1.C2`. La deuxième instance de `C2` est déclarée à l'intérieur d'un espace de noms `N2` ; par conséquent, son nom complet est : `N1.N2.C2`.

À partir du segment de code précédent, vous pouvez ajouter un nouveau membre de classe (`C3`) à l'espace de noms `N1.N2` comme ceci :

```
namespace N1.N2
{
    class C3   // N1.N2.C3
    {
    }
}
```

En règle générale, utilisez le qualificateur d'alias d'espace de noms `::` pour référencer un alias d'espace de noms ou `global::` pour référencer l'espace de noms global, et `.` pour qualifier les types ou les membres.

C'est une erreur d'utiliser `::` avec un alias qui fait référence à un type plutôt qu'à un espace de noms. Par exemple :

```
using Alias = System.Console;
```

```
class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}
```

Rappelez-vous que le mot `global` n'est pas un alias prédéfini ; par conséquent, `global.X` n'a pas de signification particulière. Il n'acquiert une signification particulière que lorsqu'il est utilisé avec `::`.

L'avertissement du compilateur CS0440 est généré si vous définissez un alias nommé global, car `global::` fait toujours référence à l'espace de noms global et non à un alias. Par exemple, la ligne suivante génère l'avertissement :

```
using global = System.Collections;   // Warning
```

Il est tout à fait opportun d'utiliser `::` avec des alias et cela vous prémunit contre l'introduction inattendue de types supplémentaires. Par exemple, prenons l'exemple suivant :

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias::Exception { }
}
```

Cela fonctionne, mais si un type nommé `Alias` devait par la suite être introduit, `Alias.` se lierait à ce type. L'utilisation de `Alias::Exception` est l'assurance que `Alias` est considéré comme un alias d'espace de noms et qu'il n'est pas pris à tort pour un type.

Voir aussi

- Guide de programmation C#
- Espaces de noms
- Expression d'accès au membre
- :: (opérateur)
- extern alias

Comment utiliser l'espace de noms My (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

L' Microsoft.VisualBasic.MyServices espace de noms (`My` dans Visual Basic) offre un accès facile et intuitif à plusieurs classes .net, ce qui vous permet d'écrire du code qui interagit avec l'ordinateur, l'application, les paramètres, les ressources, etc. Bien que conçu à l'origine pour une utilisation avec Visual Basic, l'espace de noms `MyServices` peut être utilisé dans les applications C#.

Pour plus d'informations sur l'utilisation de l'espace de noms `MyServices` de Visual Basic, consultez Développement avec My.

Ajouter une référence

Pour pouvoir utiliser les classes `MyServices` dans votre solution, vous devez d'abord ajouter une référence à la bibliothèque Visual Basic.

Ajouter une référence à la bibliothèque de Visual Basic

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nœud **Références**, puis sélectionnez **Ajouter une référence**.

2. Une fois la boîte de dialogue **Références** affichée, faites défiler la liste et sélectionnez Microsoft.VisualBasic.dll.

   Vous pouvez aussi inclure la ligne suivante de la section `using` au début de votre programme.

   ```
   using Microsoft.VisualBasic.Devices;
   ```

Exemple

Cet exemple appelle plusieurs méthodes statiques contenues dans l'espace de noms `MyServices`. Pour compiler ce code, une référence à Microsoft.VisualBasic.DLL doit être ajoutée au projet.

```
using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}
```

Certaines classes de l'espace de noms `MyServices` ne peuvent pas être appelées à partir d'une application C# : par exemple, la classe FileSystemProxy n'est pas compatible. Dans ce cas particulier, les méthodes statiques, qui font partie intégrante de FileSystem et qui sont aussi contenues dans VisualBasic.dll, peuvent être utilisées à la place. Par exemple, voici comment utiliser une méthode de ce type pour dupliquer un répertoire :

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

Voir aussi

- Guide de programmation C#
- Espaces de noms
- Utilisation d'espaces de noms

Pointeurs et code unsafe (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Pour conserver la sécurité des types, par défaut, C# ne prend pas en charge les opérations arithmétiques sur les pointeurs. Cependant, en utilisant le mot clé unsafe, vous pouvez définir un contexte non sécurisé dans lequel des pointeurs peuvent être utilisés. Pour en savoir plus sur les pointeurs, consultez la rubrique Types de pointeur.

> **NOTE**
>
> Dans le common language runtime (CLR), le code non sécurisé est appelé « code non vérifiable ». Le code non sécurisé en C# n'est pas nécessairement dangereux : il s'agit simplement de code dont la sécurité ne peut pas être vérifiée par le CLR. Le CLR exécute du code non sécurisé seulement s'il se trouve dans un assembly entièrement fiable. Si vous utilisez du code non sécurisé, il vous incombe de vérifier que votre code n'introduit pas de risques de sécurité ni d'erreurs de pointeur.

Vue d'ensemble du code unsafe

Le code non sécurisé a les propriétés suivantes :

- Les méthodes, les types et les blocs de code peuvent être définis comme non sécurisés.
- Dans certains cas, le code non sécurisé peut augmenter les performances d'une application en supprimant les vérifications des limites des tableaux.
- Du code non sécurisé est obligatoire quand vous appelez des fonctions natives nécessitant des pointeurs.
- L'utilisation de code non sécurisé introduit des risques pour la sécurité et la stabilité.
- Le code qui contient des blocs unsafe doit être compilé avec l'option de compilateur -unsafe.

Sections connexes

Pour plus d'informations, consultez :

- Types de pointeur
- Mémoires tampons de taille fixe

spécification du langage C#

Pour en savoir plus, consultez la rubrique Code unsafe de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- unsafe

Mémoires tampons de taille fixe (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

En C#, vous pouvez utiliser l'instruction fixed pour créer une mémoire tampon avec un tableau de taille fixe dans une structure de données. Les mémoires tampons de taille fixe sont utiles quand vous écrivez des méthodes qui sont interopérables avec les sources de données d'autres langages ou plateformes. Le tableau fixe peut accepter tous les attributs ou modificateurs qui sont autorisés pour les membres de structures régulières. La seule restriction est que le tableau doit être de type `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` ou `double`.

```
private fixed char name[30];
```

Notes

Dans du code safe, un struct C# qui contient un tableau ne contient pas les éléments du tableau. Le struct contient une référence aux éléments du tableau. Vous pouvez incorporer un tableau de taille fixe dans un struct quand il est utilisé dans un bloc de code unsafe.

La taille de l' `struct` élément suivant ne dépend pas du nombre d'éléments du tableau, car `pathName` est une référence :

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

Un `struct` peut contenir un tableau incorporé dans du code unsafe. Dans l'exemple suivant, le tableau `fixedBuffer` a une taille fixe. Utilisez une instruction `fixed` pour établir un pointeur vers le premier élément. Accédez aux éléments du tableau par le biais de ce pointeur. L'instruction `fixed` épingle le champ de l'instance `fixedBuffer` à un emplacement spécifique de la mémoire.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

La taille du tableau `char` de 128 éléments est de 256 octets. Les mémoires tampons char de taille fixe acceptent toujours deux octets par caractère, quel que soit l'encodage. Ceci est vrai même lorsque les mémoires tampons char sont marshalées vers des méthodes ou des structs d'API avec `CharSet = CharSet.Auto` ou `CharSet = CharSet.Ansi`. Pour plus d'informations, consultez CharSet.

L'exemple précédent montre comment accéder aux champs `fixed` sans épinglage, ce qui est possible à compter de C# 7.3.

Un autre tableau courant de taille fixe est le tableau bool. La taille des éléments d'un tableau `bool` est toujours d'un octet. Les tableaux `bool` ne conviennent pas à la création de tableaux d'octets ou de mémoires tampons.

Les mémoires tampons de taille fixe sont compilées avec le System.Runtime.CompilerServices.UnsafeValueTypeAttribute , qui indique au Common Language Runtime (CLR) qu'un type contient un tableau non managé susceptible de provoquer un dépassement de capacité. Cela est similaire à la mémoire créée à l'aide de stackalloc, qui active automatiquement les fonctionnalités de détection de dépassement de mémoire tampon dans le CLR. L'exemple précédent montre comment une mémoire tampon de taille fixe peut exister dans un `unsafe struct` .

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

Le compilateur qui a généré C# pour `Buffer` , est attribué comme suit :

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Les mémoires tampons de taille fixe diffèrent des tableaux normaux des manières suivantes :

- Peut uniquement être utilisé dans un contexte non sécurisé .
- Peut uniquement être des champs d'instance de structs.
- Il s'agit toujours de vecteurs, ou de tableaux unidimensionnels.
- La déclaration doit inclure la longueur, telle que `fixed char id[8]` . Vous ne pouvez pas utiliser `fixed char id[]`.

Voir aussi

- Guide de programmation C#
- Pointeurs et code unsafe
- fixed, instruction
- Interopérabilité

Types pointeur (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Dans un contexte unsafe, un type peut être un type pointeur, un type valeur ou un type référence. La déclaration d'un type pointeur peut prendre l'une des formes suivantes :

```
type* identifier;
void* identifier; //allowed but not recommended
```

Le type spécifié avant `*` dans un type de pointeur est appelé **type référent**. Seul un type non managé peut être un type référent.

Les types pointeur n'héritent pas de object, et aucune conversion n'est possible entre les types pointeur et `object`. Par ailleurs, le boxing et l'unboxing ne prennent pas en charge les pointeurs. Cependant, vous pouvez effectuer des conversions entre différents types pointeur ainsi qu'entre des types pointeur et des types intégraux.

Lorsque vous déclarez plusieurs pointeurs dans la même déclaration, l'astérisque (*) est écrit conjointement au type sous-jacent uniquement, il n'est pas utilisé en tant que préfixe de chaque nom de pointeur. Par exemple :

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

Un pointeur ne peut pas pointer vers une référence ou vers un struct qui contient des références, car une référence d'objet peut être collectée par le récupérateur de mémoire, même si un pointeur pointe vers elle. Le récupérateur de mémoire ne se préoccupe pas de savoir si un objet est pointé par des types pointeur.

La valeur de la variable pointeur de type `myType*` est l'adresse d'une variable de type `myType`. Les éléments suivants sont des exemples de déclarations de type pointeur :

EXEMPLE	DESCRIPTION
`int* p`	`p` est un pointeur vers un entier.
`int** p`	`p` est un pointeur vers un pointeur vers un entier.
`int*[] p`	`p` est un tableau unidimensionnel de pointeurs vers des entiers.
`char* p`	`p` est un pointeur vers un caractère.
`void* p`	`p` est un pointeur vers un type inconnu.

L'opérateur d'indirection de pointeur * peut être utilisé pour accéder au contenu à l'emplacement vers lequel pointe la variable pointeur. Observez par exemple la déclaration suivante :

```
int* myVariable;
```

L'expression `*myVariable` désigne la variable `int` trouvée à l'adresse contenue dans `myVariable`.

Plusieurs exemples de pointeurs sont présentés dans les rubriques fixed, instruction et Conversions de pointeur. L'exemple suivant utilise le mot clé `unsafe` et les instructions `fixed`, et montre comment incrémenter un pointeur intérieur. Vous pouvez coller ce code dans la fonction Main d'une application console pour l'exécuter. Ces exemples doivent être compilés avec l'ensemble d'options de compilateur -unsafe.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

L'opérateur d'indirection ne peut pas être appliqué à un pointeur de type `void*`. Toutefois, vous pouvez utiliser un cast pour convertir un pointeur void en n'importe quel autre type pointeur, et inversement.

Un pointeur peut être `null`. Le fait d'appliquer un opérateur d'indirection à un pointeur Null donne lieu à un comportement défini par l'implémentation.

Le passage de pointeurs entre méthodes peut engendrer un comportement non défini. Supposons une méthode qui retourne un pointeur à une variable locale par le biais d'un paramètre `in`, `out` ou `ref`, ou comme résultat de fonction. Si le pointeur a été défini dans un bloc fixed, la variable vers laquelle il pointe peut ne plus être fixed.

Le tableau suivant répertorie les opérateurs et les instructions qui peuvent fonctionner sur des pointeurs dans un contexte unsafe :

OPÉRATEUR/INSTRUCTION	UTILISATION
`*`	Exécute l'indirection de pointeur.
`->`	Accède à un membre d'un struct via un pointeur.
`[]`	Indexe un pointeur.
`&`	Obtient l'adresse d'une variable.
`++` et `--`	Incrémente et décrémente les pointeurs.
`+` et `-`	Exécute des opérations arithmétiques sur les pointeurs.
`==`, `!=`, `<`, `>`, `<=` et `>=`	Compare des pointeurs.
`stackalloc`	Alloue de la mémoire sur la pile.
`fixed` gestion	Résout temporairement une variable afin de pouvoir rechercher son adresse.

Pour plus d'informations sur les opérateurs associés au pointeur, consultez Opérateurs associés au pointeur.

spécification du langage C#

Pour plus d'informations, consultez la section Types de pointeur de la Spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Pointeurs et code unsafe
- Conversions de pointeur
- Types référence
- Types de valeur
- unsafe

Conversions de pointeur (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Le tableau suivant présente les conversions de pointeur implicites prédéfinies. Les conversions implicites peuvent se produire dans de nombreuses situations, comme les appels de méthode et les instructions d'assignation.

Conversions de pointeur implicites

À PARTIR	À
Tout type pointeur	void*
null	Tout type pointeur

La conversion de pointeur explicite permet d'effectuer des conversions, pour lesquelles il n'existe pas de conversion implicite, à l'aide d'une expression de cast. Le tableau suivant liste ces conversions.

Conversions de pointeur explicites

À PARTIR	À
Tout type pointeur	Tout autre type pointeur
sbyte, byte, short, ushort, int, uint, long ou ulong	Tout type pointeur
Tout type pointeur	sbyte, byte, short, ushort, int, uint, long ou ulong

Exemple

Dans l'exemple suivant, un pointeur vers `int` est converti en pointeur vers `byte`. Notez que le pointeur pointe vers l'octet traité le plus faible de la variable. Quand vous incrémentez successivement le résultat, jusqu'à la taille de `int` (4 octets), vous pouvez afficher les octets restants de la variable.

```
// compile with: -unsafe
```

```
class ClassConvert
{
    static void Main()
    {
        int number = 1024;

        unsafe
        {
            // Convert to byte:
            byte* p = (byte*)&number;

            System.Console.Write("The 4 bytes of the integer:");

            // Display the 4 bytes of the int variable:
            for (int i = 0 ; i < sizeof(int) ; ++i)
            {
                System.Console.Write(" {0:X2}", *p);
                // Increment the pointer:
                p++;
            }
            System.Console.WriteLine();
            System.Console.WriteLine("The value of the integer: {0}", number);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
```

Voir aussi

- Guide de programmation C#
- Types de pointeur
- Types référence
- Types de valeur
- unsafe
- fixed, instruction
- stackalloc

Comment utiliser des pointeurs pour copier un tableau d'octets (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

L'exemple suivant utilise des pointeurs pour copier des octets d'un tableau à un autre.

Cet exemple utilise le mot clé unsafe, qui vous permet d'utiliser des pointeurs dans la méthode `Copy`. L'instruction fixed permet de déclarer des pointeurs vers les tableaux source et de destination. L'instruction `fixed` *épingle* l'emplacement des tableaux source et de destination en mémoire pour que ceux-ci ne soient pas déplacés par l'opération de garbage collection. Les blocs de mémoire des tableaux sont libérés quand le bloc `fixed` est effectué. Comme la méthode `Copy` dans cet exemple utilise le mot clé `unsafe`, elle doit être compilée avec l'option de compilateur -unsafe.

Cet exemple accède aux éléments des deux tableaux au moyen d'index et non au moyen d'un second pointeur non managé. La déclaration des pointeurs `pSource` et `pTarget` épingle les tableaux. Cette fonctionnalité est disponible à compter de C# 7.3.

Exemple

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}
```

```
static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

Voir aussi

- Guide de programmation C#
- Pointeurs et code unsafe
- -unsafe (Options du compilateur C#)
- Garbage collection

Commentaires de documentation XML (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

En C#, vous pouvez créer de la documentation pour votre code en incluant des éléments XML dans des champs de commentaire spéciaux (indiqués par trois barres obliques) dans le code source, juste avant le bloc de code auquel les commentaires font référence, par exemple.

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

Quand vous compilez avec l'option -doc , le compilateur recherche toutes les balises XML dans le code source et crée un fichier de documentation XML. Pour créer la documentation finale basée sur le fichier généré par le compilateur, vous pouvez créer un outil personnalisé ou utiliser un outil tel que DocFX ou Sandcastle.

Pour faire référence à des éléments XML (par exemple, votre fonction traite des éléments XML spécifiques que vous souhaitez décrire dans un commentaire de documentation XML), vous pouvez utiliser le mécanisme de citation standard (`<` et `>`). Pour faire référence aux identificateurs génériques dans les éléments de référence de code (`cref`), vous pouvez utiliser des caractères d'échappement (par exemple, `cref="List<T>"`) ou des accolades (`cref="List{T}"`). En tant que cas particulier, le compilateur analyse les accolades comme des crochets pointus pour rendre le commentaire de documentation moins fastidieux à créer lorsqu'il s'agit de faire référence aux identificateurs génériques.

> **NOTE**
>
> Les commentaires de documentation XML ne sont pas des métadonnées. Ils ne sont pas inclus dans l'assembly compilé et ne sont donc pas accessibles par réflexion.

Contenu de cette section

- Balises recommandées pour les commentaires de documentation
- Traitement du fichier XML
- Délimiteurs pour les balises de documentation
- Comment utiliser les fonctionnalités de la documentation XML

Sections connexes

Pour plus d'informations, consultez :

- -doc (traiter les commentaires de documentation)

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#

Balises recommandées pour les commentaires de documentation (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Le compilateur C# traite les commentaires de documentation dans votre code et les met au format XML dans un fichier dont vous spécifiez le nom dans l'option de ligne de commande **/doc**. Pour créer la documentation finale basée sur le fichier généré par le compilateur, vous pouvez créer un outil personnalisé ou utiliser un outil tel que DocFX ou Sandcastle.

Les balises sont traitées sur les constructions de code telles que les types et les membres de types.

> **NOTE**
> Il n'est pas possible d'appliquer des commentaires de documentation à un espace de noms.

Le compilateur traite toute balise représentant du code XML correct. Les balises suivantes fournissent des fonctionnalités fréquemment utilisées dans la documentation utilisateur.

Balises

<c>	<para>	<see>*	<value>
<code>	<param>*	<seealso>*	
<example>	<paramref>	<summary>	
<exception>*	<permission>*	<typeparam>*	
<include>*	<remarks>	<typeparamref>	
<list>	<inheritdoc>	<returns>	

(* indique que le compilateur vérifie la syntaxe.)

Si vous voulez que des crochets pointus apparaissent dans le texte d'un commentaire de documentation, utilisez le codage HTML de `<` et `>`, respectivement `<` et `>`. Cet encodage est illustré dans l'exemple suivant.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

Voir aussi

- Guide de programmation C#
- -doc (options du compilateur C#)
- Commentaires sur la documentation XML

Traiter le fichier XML (Guide de programmation C#)

02/11/2020 • 10 minutes to read • Edit Online

Le compilateur génère une chaîne d'ID pour chaque construction de votre code balisée pour générer la documentation. (Pour plus d'informations sur la façon de baliser votre code, consultez Balises recommandées pour les commentaires de documentation.) La chaîne d'ID identifie de façon unique la construction. Les programmes qui traitent le fichier XML peuvent utiliser la chaîne d'ID pour identifier les métadonnées .NET ou l'élément de réflexion correspondants auxquels la documentation s'applique.

Chaînes d'ID

Le fichier XML n'est pas une représentation hiérarchique de votre code. Il s'agit d'une liste plate qui a un ID généré pour chaque élément.

Le compilateur respecte les règles suivantes quand il génère les chaînes d'ID :

- La chaîne ne contient aucun espace blanc.
- La première partie de la chaîne identifie le type de membre à l'aide d'un caractère unique suivi d'un signe deux-points. Les types de membres suivants sont utilisés :

CARACTÈRE	TYPE DE MEMBRE	NOTES
N	espace de noms	Vous ne pouvez pas ajouter de commentaires de documentation à un espace de noms, mais vous pouvez faire des références cref à des commentaires, si cela est pris en charge.
T	type	Un type peut être une classe, une interface, un struct, une énumération ou un délégué.
F	field	
P	propriété	Comprend des indexeurs ou d'autres propriétés indexées.
M	method	Comprend des méthodes spéciales, telles que des constructeurs et des opérateurs.
E	événement	
!	chaîne d'erreur	Le reste de la chaîne fournit des informations sur l'erreur. Le compilateur C# génère des informations d'erreur pour les liens qui ne peuvent pas être résolus.

- La deuxième partie de la chaîne est le nom qualifié complet de l'élément, en commençant à la racine de l'espace de noms. Le nom de l'élément, ses types englobants et l'espace de noms sont séparés par des

points. Si le nom de l'élément lui-même comporte des points, ceux-ci sont remplacés par un signe dièse (« # »). Il est supposé qu'aucun élément n'a de signature de hachage directement dans son nom. Par exemple, le nom qualifié complet du constructeur de chaîne est « System. String. #ctor ».

- Pour les propriétés et les méthodes, la liste de paramètres entre parenthèses suit. S'il n'y a aucun paramètre, aucune parenthèse n'est présente. Les paramètres sont séparés par des virgules. L'encodage de chaque paramètre suit directement la manière dont il est encodé dans une signature .NET :
  - Types de base. Les types réguliers (ELEMENT_TYPE_CLASS ou ELEMENT_TYPE_VALUETYPE) sont représentés en tant que nom qualifié complet du type.
  - Les types intrinsèques (par exemple, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF et ELEMENT_TYPE_VOID) sont représentés en tant que nom qualifié complet du type complet correspondant. Par exemple, System.Int32 ou System.TypedReference.
  - ELEMENT_TYPE_PTR est représenté par un « * » après le type modifié.
  - ELEMENT_TYPE_BYREF est représenté par un '@' après le type modifié.
  - ELEMENT_TYPE_PINNED est représenté par un « ^ » après le type modifié. Le compilateur C# ne génère jamais ceci.
  - ELEMENT_TYPE_CMOD_REQ est représenté par un « | » et le nom qualifié complet de la classe de modification, après le type modifié. Le compilateur C# ne génère jamais ceci.
  - ELEMENT_TYPE_CMOD_OPT est représenté par un « ! » et le nom qualifié complet de la classe de modification, après le type modifié.
  - ELEMENT_TYPE_SZARRAY est représenté par « [] » après le type d'élément du tableau.
  - ELEMENT_TYPE_GENERICARRAY est représenté par « [?] » après le type d'élément du tableau. Le compilateur C# ne génère jamais ceci.
  - ELEMENT_TYPE_ARRAY est représenté sous la*forme [Lower*: `size` , Lower :*lowerbound* `size`], où le nombre de virgules est le rang-1, et les limites inférieures et la taille de chaque dimension, si elles sont connues, sont représentées au format décimal. Si une limite inférieure ou une taille n'est pas spécifiée, elle est omise. Si la limite inférieure et la taille d'une dimension particulière sont omises, le « : » est également omis. Par exemple, un tableau à deux dimensions avec 1 comme limite inférieure et une taille non spécifiée est [1:,1:].
  - ELEMENT_TYPE_FNPTR est représenté en tant que « =FUNC: `type` (*signature*) », où `type` est le type de retour et *signature* correspond aux arguments de la méthode. S'il n'y a pas d'argument, les parenthèses sont omises. Le compilateur C# ne génère jamais ceci.

  Les composants de signature suivants ne sont pas représentés, car ils ne sont pas utilisés pour différencier les méthodes surchargées :
  - convention d'appel
  - type de retour
  - ELEMENT_TYPE_SENTINEL
- Pour les opérateurs de conversion uniquement (`op_Implicit` et `op_Explicit`), la valeur de retour de la méthode est encodée sous la forme d'un « ~ » suivi du type de retour.
- Pour les types génériques, le nom du type est suivi d'un accent grave, puis d'un chiffre qui indique le nombre de paramètres de type générique. Par exemple :

`<member name="T:SampleClass`2">` est l'étiquette pour un type qui est défini en tant que `public class SampleClass<T, U>`.

Pour les méthodes qui prennent des types génériques comme paramètres, les paramètres de type générique sont spécifiés sous la forme de nombres précédés d'impulsions (par exemple `, 0, ` 1). Chaque nombre représente une notation de tableau de base zéro pour les paramètres génériques du type.

Exemples

Les exemples suivants montrent comment les chaînes d'ID pour une classe et ses membres sont générées :

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public X(int i) { }

        /// <summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int f() { return 1; }

        /// <summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <param name="y">Describe parameter.</param>
        /// <param name="z">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }

        /// <summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
```

```
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Addition(N.X,N.X)".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <param name="xx">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static X operator +(X x, X xx) { return x; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }

        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:N.X.d".
        /// </summary>
        public event D d;

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.Item(System.String)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:N.X.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:N.X.D".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void D(int i);

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(X x) { return 1; }
    }
}
```

Voir aussi

- Guide de programmation C#
- -doc (options du compilateur C#)
- Commentaires sur la documentation XML

Délimiteurs pour les balises de documentation (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

L'utilisation de commentaires de documentation XML exige des délimiteurs, qui indiquent au compilateur où un commentaire de documentation commence et se termine. Vous pouvez utiliser les genres de délimiteurs ci-dessous avec les balises de documentation XML :

- `///`

  Délimiteur à une ligne. Il s'agit de la forme qui est présentée dans les exemples de documentation et utilisée par les modèles de projet C#. Si un caractère espace blanc suit le délimiteur, ce caractère n'est pas inclus dans la sortie XML.

  > **NOTE**
  > L'environnement de développement intégré (IDE) de Visual Studio insère automatiquement `<summary>` les `</summary>` balises et et déplace le curseur dans ces balises après que vous avez tapé le `///` délimiteur dans l'éditeur de code. Vous pouvez activer/désactiver cette fonctionnalité dans la boîte de dialogue Options.

- `/** */`

  Séparateurs multilignes.

  Certaines règles de mise en forme doivent être respectées lorsque vous utilisez les `/** */` délimiteurs :

  - Sur la ligne qui contient le délimiteur `/**`, si le reste de la ligne n'est constitué que d'espace blanc, la ligne n'est pas traitée pour les commentaires. Si le premier caractère après le délimiteur `/**` est un espace blanc, celui-ci est ignoré et le reste de la ligne est traité. Sinon, tout le texte de la ligne après le délimiteur `/**` est traité comme faisant partie du commentaire.
  - Sur la ligne qui contient le délimiteur `*/`, s'il n'y a que des espaces blancs jusqu'au délimiteur `*/`, cette ligne est ignorée. Dans le cas contraire, le texte sur la ligne jusqu'au `*/` délimiteur est traité dans le cadre du commentaire.
  - Pour les lignes situées après celle qui commence par le délimiteur `/**`, le compilateur recherche un modèle commun au début de chaque ligne. Le modèle peut se composer d'un espace blanc facultatif et d'un astérisque (`*`), suivi d'autres espaces blancs facultatifs. Si le compilateur détecte un modèle commun au début de chaque ligne qui ne commence pas par le délimiteur `/**` ou `*/`, il ignore ce modèle pour chaque ligne.

  Les exemples suivants illustrent ces règles.

  - La seule partie du commentaire suivant traitée est la ligne qui commence par `<summary>` . Les trois formats de balise produisent les mêmes commentaires.

```
/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
* <summary>text</summary>
*/
```

- Le compilateur identifie un modèle commun de « * » au début des deuxième et troisième lignes. Le modèle n'est pas inclus dans la sortie.

```
/**
* <summary>
* text </summary>*/
```

- Le compilateur ne détecte aucun modèle commun dans le commentaire suivant, car le deuxième caractère sur la troisième ligne n'est pas un astérisque. Ainsi, tout le texte des deuxième et troisième lignes est traité comme faisant partie du commentaire.

```
/**
* <summary>
   text </summary>
*/
```

- Le compilateur ne détecte aucun modèle dans le commentaire suivant pour deux raisons. Tout d'abord, le nombre d'espaces avant l'astérisque n'est pas cohérent. Ensuite, la cinquième ligne commence par une tabulation, qui ne correspond pas à des espaces. Ainsi, tout le texte des lignes deux à cinq est traité comme faisant partie du commentaire.

```
/**
  * <summary>
  * text
*  text2
    *  </summary>
*/
```

Voir aussi

- Guide de programmation C#
- Commentaires sur la documentation XML
- -doc (options du compilateur C#)

Comment utiliser les fonctionnalités de la documentation XML

02/11/2020 • 5 minutes to read • Edit Online

L'exemple suivant montre un type qui a été documenté.

Exemple

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
```

```
    }

    /// <summary>
    /// Some other method.
    /// </summary>
    /// <returns>
    /// Return values are described through the returns tag.
    /// </returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific method.
    /// </seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }

    public int InterfaceMethod(int n)
    {
        return n * n;
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments.</param>
    static int Main(System.String[] args)
    {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}
```

L'exemple génère un fichier *.xml* avec le contenu suivant.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlsample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
            Class level summary documentation goes here.
            </summary>
            <remarks>
            Longer comments can be associated with a type or member through
            the remarks tag.
```

```
            </remarks>
        </member>
        <member name="F:TestClass._name">
            <summary>
            Store for the Name property.
            </summary>
        </member>
        <member name="M:TestClass.#ctor">
            <summary>
            The class constructor.
            </summary>
        </member>
        <member name="P:TestClass.Name">
            <summary>
            Name property.
            </summary>
            <value>
            A value tag is used to describe the property value.
            </value>
        </member>
        <member name="M:TestClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.
            </summary>
            <param name="s"> Parameter description for s goes here.</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference a type or member
            and the compiler will check that the reference exists.
            </seealso>
        </member>
        <member name="M:TestClass.SomeOtherMethod">
            <summary>
            Some other method.
            </summary>
            <returns>
            Return values are described through the returns tag.
            </returns>
            <seealso cref="M:TestClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a specific method.
            </seealso>
        </member>
        <member name="M:TestClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments.</param>
        </member>
        <member name="T:TestInterface">
            <summary>
            Documentation that describes the interface goes here.
            </summary>
            <remarks>
            Details about the interface go here.
            </remarks>
        </member>
        <member name="M:TestInterface.InterfaceMethod(System.Int32)">
            <summary>
            Documentation that describes the method goes here.
            </summary>
            <param name="n">
            Parameter n requires an integer argument.
            </param>
            <returns>
            The method returns an integer.
            </returns>
        </member>
    </members>
</doc>
```

Compilation du code

Pour compiler l'exemple, entrez la commande suivante :

```
csc XMLsample.cs /doc:XMLsample.xml
```

Cette commande crée le fichier XML *XMLsample.xml*, que vous pouvez afficher dans votre navigateur ou à l'aide de la `TYPE` commande.

Programmation fiable

La documentation XML commence par `///` . Lorsque vous créez un projet, les assistants placent des lignes de démarrage `///` pour vous. Le traitement de ces commentaires présente certaines restrictions :

- La documentation doit être dans un format XML correct. Si le XML n'est pas correct, un avertissement est généré. En outre, un commentaire indiquant qu'une erreur s'est produite est ajouté au fichier de documentation.
- Les développeurs sont libres de créer leur propre jeu de balises. Il existe un ensemble recommandé de balises. Certaines des balises recommandées ont des significations spéciales :
  - La `<param>` balise est utilisée pour décrire les paramètres. Quand elle est utilisée, le compilateur vérifie que le paramètre existe et que tous les paramètres sont décrits dans la documentation. Si la vérification échoue, le compilateur émet un avertissement.
  - L' `cref` attribut peut être attaché à n'importe quelle balise pour référencer un élément de code. Le compilateur vérifie l'existence de cet élément de code. Si la vérification échoue, le compilateur émet un avertissement. Le compilateur respecte toutes les instructions `using` lorsqu'il recherche un type décrit dans l'attribut `cref` .
  - La `<summary>` balise est utilisée par IntelliSense dans Visual Studio pour afficher des informations supplémentaires sur un type ou un membre.

    > **NOTE**
    > Le fichier XML ne fournit pas des informations complètes sur le type et les membres (par exemple, il ne contient pas d'informations sur le type). Pour obtenir des informations complètes sur un type ou un membre, utilisez le fichier de documentation ainsi que la réflexion sur le type ou le membre réel.

Voir aussi

- Guide de programmation C#
- -doc (options du compilateur C#)
- Commentaires sur la documentation XML
- Processeur de documentation DocFX
- Processeur de documentation Sandcastle

<c>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<c>text</c>
```

Paramètres

- `text`

  Texte que vous souhaitez indiquer comme étant du code.

Notes

La `<c>` balise vous donne un moyen d'indiquer que le texte d'une description doit être marqué comme étant du code. Utilisez <code> pour indiquer plusieurs lignes en tant que code.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<code>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<code>content</code>
```

Paramètres

- `content`

  Texte à marquer comme étant du code.

Notes

La `<code>` balise est utilisée pour indiquer plusieurs lignes de code. Utilisez <c> pour indiquer que le texte d'une seule ligne dans une description doit être marqué comme étant du code.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

Consultez l' <example> article pour obtenir un exemple d'utilisation de la `<code>` balise.

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

CREF, attribut (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

L'attribut `cref` dans une balise de documentation XML signifie « référence de code ». Il indique que le texte interne de la balise est un élément de code tel qu'un type, une méthode ou une propriété. Les outils de documentation comme DocFX et Sandcastle utilisent les attributs `cref` pour générer automatiquement des liens hypertexte vers la page où le type ou le membre est documenté.

Exemple

L'exemple suivant montre les `cref` attributs utilisés dans les `<see>` balises.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>
```

```
        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

Une fois compilé, le programme produit le fichier XML suivant. Notez que l'attribut `cref` pour la méthode `GetZero`, par exemple, a été transformé par le compilateur en `"M:TestNamespace.TestClass.GetZero"`. Le préfixe « M: » signifie « méthode » et représente une convention reconnue par les outils de documentation tels que DocFX et Sandcastle. Pour obtenir une liste complète de préfixes, consultez Traitement du fichier XML.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains several cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
            This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref
attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
            This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/>
constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example>
            This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>
</doc>
```

Voir aussi

- Commentaires sur la documentation XML
- Balises recommandées pour les commentaires de documentation

<example>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<example>description</example>
```

Paramètres

- `description`

  Description de l'exemple de code.

Notes

La `<example>` balise vous permet de spécifier un exemple d'utilisation d'une méthode ou d'un autre membre de bibliothèque. Cela implique généralement l'utilisation de la <code> balise.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
```

```
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<exception>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<exception cref="member">description</exception>
```

Paramètres

- cref = " `member` "

  Référence à une exception qui est disponible à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'exception donnée existe, et il traduit `member` en nom d'élément canonique dans le fichier XML de sortie. `member` doit apparaître entre guillemets doubles (" ").

  Pour plus d'informations sur la façon de mettre en forme `member` pour référencer un type générique, consultez Traitement du fichier XML.

- `description`

  Description de l'exception.

Notes

La `<exception>` balise vous permet de spécifier les exceptions qui peuvent être levées. Cette balise peut être appliquée à des définitions de méthodes, de propriétés, d'événements et d'indexeurs.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Pour plus d'informations sur la gestion des exceptions, consultez Exceptions et gestion des exceptions.

Exemple

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<include> (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Syntaxe

```
<include file='filename' path='tagpath[@name="id"]' />
```

Paramètres

- `filename`

  Nom du fichier XML contenant la documentation. Le nom de fichier peut être qualifié avec un chemin relatif au fichier de code source. Mettez `filename` entre guillemets simples (' ').

- `tagpath`

  Chemin des balises contenues dans `filename` qui mène à la balise `name`. Mettez le chemin entre guillemets simples (' ').

- `name`

  Spécificateur de nom contenu dans la balise qui précède les commentaires ; `name` possède un `id`.

- `id`

  ID de la balise qui précède les commentaires. Mettez l'ID entre guillemets doubles (" ").

Remarques

La `<include>` balise vous permet de faire référence aux commentaires d'un autre fichier qui décrivent les types et les membres de votre code source. Il s'agit d'une solution alternative au placement direct des commentaires de la documentation dans votre fichier de code source. En plaçant la documentation dans un fichier distinct, vous pouvez appliquer un contrôle de code source à la documentation indépendamment du code source. Ainsi, une personne peut extraire le fichier de code source et une autre personne peut extraire le fichier de documentation.

La `<include>` balise utilise la syntaxe XML XPath. Reportez-vous à la documentation XPath pour savoir comment personnaliser votre `<include>` utilisation.

Exemple

Cet exemple comprend plusieurs fichiers. Voici le premier fichier, qui utilise `<include>` .

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

Le deuxième fichier, *xml_include_tag.doc*, contient les commentaires de documentation suivants.

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Sortie du programme

La sortie suivante est générée quand vous compilez les classes Test et Test2 avec la ligne de commande suivante : `-doc:DocFileName.xml.` Dans Visual Studio, l'option de commentaires de document XML est spécifiée dans le volet de génération du Concepteur de projet. Lorsque le compilateur C# voit la `<include>` balise, il recherche des commentaires de documentation dans *xml_include_tag.doc* au lieu du fichier source actuel. Le compilateur génère ensuite *DocFileName.xml*, et il s'agit du fichier utilisé par les outils de documentation tels que Sandcastle pour produire la documentation finale.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<inheritdoc> (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<inheritdoc/>
```

InheritDoc

Hériter des commentaires XML à partir des classes de base, des interfaces et des méthodes similaires. Cela élimine la copie et le collage indésirables de commentaires XML en double et conserve automatiquement les commentaires XML synchronisés.

Notes

Ajoutez vos commentaires XML dans des classes ou des interfaces de base et laissez InheritDoc copier les commentaires vers les classes d'implémentation.

Ajoutez vos commentaires XML à vos méthodes synchrones et laissez InheritDoc copier les commentaires dans vos versions asynchrones des mêmes méthodes.

Si vous souhaitez copier les commentaires à partir d'un membre spécifique, vous pouvez utiliser l' `cref` attribut pour spécifier le membre.

Exemples

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
public class MainClass
{
}

///<inheritdoc/>
public class TestClass: MainClass
{
}
```

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this interface.
/// </summary>
public interface ITestInterface
{
}

///<inheritdoc cref="ITestInterface"/>
public class TestClass : ITestInterface
{
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<list> (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>term</term>
        <description>description</description>
    </item>
</list>
```

Paramètres

- `term`

  Terme à définir, qui est défini dans `description`.

- `description`

  Élément contenu dans une puce ou une liste numérotée ou définition d'un `term`.

Notes

Le `<listheader>` bloc est utilisé pour définir la ligne d'en-tête d'une table ou d'une liste de définitions. Au moment de définir une table, il vous suffit de fournir une entrée pour le terme figurant dans l'en-tête.

Chaque élément de la liste est spécifié avec un `<item>` bloc. Au moment de créer une liste de définitions, vous devez spécifier à la fois `term` et `description`. Cependant, pour une table, une liste à puces ou une liste numérotée, il vous suffit de fournir une entrée pour `description`.

Une liste ou une table peut avoir autant de `<item>` blocs que nécessaire.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<para>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<para>content</para>
```

Paramètres

- `content`

  Texte du paragraphe.

Notes

La `<para>` balise est destinée à être utilisée à l'intérieur d'une balise, telle que <summary> , <remarks> ou <returns> , et vous permet d'ajouter une structure au texte.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

<summary>Pour obtenir un exemple d'utilisation de `<para>` , consultez.

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<param>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<param name="name">description</param>
```

Paramètres

- `name`

  Nom d'un paramètre de méthode. Mettez le nom entre guillemets doubles (" ").

- `description`

  Description du paramètre.

Notes

La `<param>` balise doit être utilisée dans le commentaire d'une déclaration de méthode pour décrire l'un des paramètres de la méthode. Pour documenter plusieurs paramètres, utilisez plusieurs `<param>` balises.

Le texte de la `<param>` balise s'affiche dans IntelliSense, dans l'Explorateur d'objets et dans le rapport Web de commentaire de code.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<paramref>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<paramref name="name"/>
```

Paramètres

- `name`

  Nom du paramètre auquel faire référence. Mettez le nom entre guillemets doubles (" ").

Notes

La `<paramref>` balise vous donne un moyen d'indiquer qu'un mot dans les commentaires de code, par exemple dans un `<summary>` `<remarks>` bloc ou fait référence à un paramètre. Le fichier XML peut être traité de manière à mettre en forme ce mot d'une façon particulière, par exemple en gras ou en italique.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<permission>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<permission cref="member">description</permission>
```

Paramètres

- cref = " `member` "

  Référence à un membre ou un champ qu'il est possible d'appeler à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'élément de code donné existe et traduit `member` en nom d'élément canonique dans le fichier XML de sortie. Le *membre* doit apparaître entre guillemets doubles (" ").

  Pour plus d'informations sur la création d'une référence cref à un type générique, consultez attribut cref.

- `description`

  Description de l'accès au membre.

Notes

La `<permission>` balise vous permet de documenter l'accès d'un membre. La classe PermissionSet vous permet de spécifier l'accès à un membre.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<remarks>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<remarks>description</remarks>
```

Paramètres

- `Description`

  Description du membre.

Notes

La `<remarks>` balise est utilisée pour ajouter des informations sur un type et compléter les informations spécifiées par <summary> . Ces informations sont affichées dans la fenêtre Explorateur d'objets.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<returns>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<returns>description</returns>
```

Paramètres

- `description`

  Description de la valeur de retour.

Notes

La `<returns>` balise doit être utilisée dans le commentaire pour une déclaration de méthode afin de décrire la valeur de retour.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<see>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<see cref="member"/>
```

Paramètres

- CREF = " `member` "

  Référence à un membre ou un champ qu'il est possible d'appeler à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'élément de code donné existe, et qu'il passe `member` au nom d'élément dans le code XML de sortie. Placez le *membre* entre guillemets doubles (" ").

Notes

La `<see>` balise vous permet de spécifier un lien à partir de texte. Utilisez <seealso> pour indiquer que le texte doit être placé dans une section Voir aussi. Utilisez l'attribut cref pour créer des liens hypertexte internes aux pages de documentation pour les éléments de code. En outre, `href` est un attribut valide qui fonctionnera comme un lien hypertexte.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

L'exemple suivant montre une `<see>` balise dans une section Résumé.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<seealso>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<seealso cref="member"/>
```

Paramètres

- cref = " `member` "

  Référence à un membre ou un champ qu'il est possible d'appeler à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'élément de code donné existe, et qu'il passe `member` au nom d'élément dans le code XML de sortie. `member` doit être placé entre guillemets doubles (" ").

  Pour plus d'informations sur la création d'une référence cref à un type générique, consultez attribut cref.

Notes

La `<seealso>` balise vous permet de spécifier le texte que vous souhaitez voir apparaître dans une section Voir aussi. Utilisez <see> pour spécifier un lien à partir de texte.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

<summary>Pour obtenir un exemple d'utilisation de <seealso> , consultez.

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<summary>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<summary>description</summary>
```

Paramètres

- `description`

  Résumé de l'objet.

Notes

La `<summary>` balise doit être utilisée pour décrire un type ou un membre de type. Utilisez <remarks> pour ajouter des informations supplémentaires à une description de type. Utilisez l'attribut cref pour activer des outils de documentation tels que DocFX et Sandcastle afin de créer des liens hypertexte internes aux pages de documentation pour les éléments de code.

Le texte de la `<summary>` balise est la seule source d'informations sur le type dans IntelliSense et s'affiche également dans la fenêtre de l'Explorateur d'objets.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter. Pour créer la documentation finale basée sur le fichier généré par le compilateur, vous pouvez créer un outil personnalisé ou utiliser un outil tel que DocFX ou Sandcastle.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

L'exemple précédent génère le fichier XML suivant.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            text for class TestClass
        </member>
        <member name="M:TestClass.DoWork(System.Int32)">
            <summary>DoWork is a method in the TestClass class.
            <para>Here's how you could make a second paragraph in a description. <see
cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
            </summary>
            <seealso cref="M:TestClass.Main"/>
        </member>
        <member name="M:TestClass.Main">
            text for Main
        </member>
    </members>
</doc>
```

Exemple

L'exemple suivant montre comment effectuer une référence `cref` à un type générique.

```
// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }

class Program
{
    static void Main() { }
}
```

L'exemple précédent génère le fichier XML suivant.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:A">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:B">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:C`1">
            <summary cref="T:A">
            </summary>
            <typeparam name="T"></typeparam>
        </member>
    </members>
</doc>
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<typeparam>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<typeparam name="name">description</typeparam>
```

Paramètres

- `name`

  Nom du paramètre de type. Mettez le nom entre guillemets doubles (" ").

- `description`

  Description du paramètre de type.

Notes

La balise `<typeparam>` doit être utilisée dans le commentaire d'une déclaration de méthode ou de type générique pour décrire un paramètre de type. Ajoutez une balise pour chaque paramètre de type de la méthode et du type générique.

Pour plus d'informations, consultez Génériques.

Le texte de la balise `<typeparam>` s'affiche dans IntelliSense, dans la fenêtre Explorateur d'objets et dans le rapport web de commentaire de code.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

Voir aussi

- Informations de référence sur C#
- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<typeparamref>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<typeparamref name="name"/>
```

Paramètres

- `name`

  Nom du paramètre de type. Mettez le nom entre guillemets doubles (" ").

Notes

Pour plus d'informations sur les paramètres de type dans les types et méthodes génériques, consultez Génériques.

Utilisez cette balise pour permettre aux consommateurs du fichier de documentation d'appliquer une mise en forme particulière au mot, par exemple l'italique.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

<value>(Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Syntaxe

```
<value>property-description</value>
```

Paramètres

- `property-description`

  Description de la propriété.

Notes

La `<value>` balise vous permet de décrire la valeur représentée par une propriété. Lorsque vous ajoutez une propriété à l'aide de l'Assistant code dans l'environnement de développement Visual Studio .NET, elle ajoute une <summary> balise pour la nouvelle propriété. Vous devez ensuite ajouter manuellement une `<value>` balise pour décrire la valeur représentée par la propriété.

Compilez avec -doc pour placer les commentaires de documentation dans un fichier en vue de les traiter.

Exemple

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Balises recommandées pour les commentaires de documentation

Exceptions et gestion des exceptions (Guide de programmation C#)

16/12/2020 • 5 minutes to read • Edit Online

Les fonctionnalités de gestion des exceptions du langage C# vous aident à gérer les situations inattendues ou exceptionnelles qui se produisent lorsqu'un programme est en cours d'exécution. La gestion des exceptions utilise les `try` `catch` `finally` Mots clés, et pour essayer des actions qui peuvent échouer, pour gérer les défaillances lorsque vous décidez qu'il est raisonnable de le faire et pour nettoyer les ressources par la suite. Les exceptions peuvent être générées par le common language runtime (CLR), par les bibliothèques .NET ou tierces, ou par le code d'application. Les exceptions sont créées avec le mot clé `throw`.

Dans de nombreux cas, une exception peut être levée non pas par une méthode appelée directement par votre code, mais par une autre méthode plus loin dans la pile des appels. Quand une exception est levée, le CLR déroulera la pile, en recherchant une méthode avec un `catch` bloc pour le type d'exception spécifique, et exécutera le premier bloc de ce type `catch` que if trouve. S'il ne trouve pas de bloc `catch` dans la pile des appels, il termine le processus et affiche un message à l'utilisateur.

Dans cet exemple, une méthode teste la division par zéro et intercepte l'erreur. Sans la gestion des exceptions, ce programme se terminerait avec une erreur **DivideByZeroException non gérée**.

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Vue d'ensemble des exceptions

Les exceptions ont les propriétés suivantes :

- Les exceptions sont des types qui dérivent tous en définitive de `System.Exception`.
- Utilisez un bloc `try` autour des instructions qui peuvent lever des exceptions.

- Dès qu'une exception se produit dans le bloc `try`, le flux de contrôle passe immédiatement au premier gestionnaire d'exceptions associé présent dans la pile des appels. En C#, le mot clé `catch` est utilisé pour définir un gestionnaire d'exceptions.
- Si aucun gestionnaire d'exceptions n'est présent pour une exception donnée, le programme s'arrête avec un message d'erreur.
- N'interceptez pas d'exception, sauf si vous pouvez la gérer et conserver l'application dans un état connu. Si vous interceptez `System.Exception`, levez-la de nouveau avec le mot clé `throw` à la fin du bloc `catch`.
- Si un bloc `catch` définit une variable d'exception, vous pouvez l'utiliser pour obtenir plus d'informations sur le type d'exception qui s'est produit.
- Les exceptions peuvent être générées explicitement par un programme avec le mot clé `throw`.
- Les objets Exception contiennent des informations détaillées sur l'erreur, telles que l'état de la pile des appels et une description du texte de l'erreur.
- Le code qui se trouve dans un bloc `finally` est exécuté même si une exception est levée. Utilisez un bloc `finally` pour libérer des ressources, par exemple pour fermer tous les flux ou fichiers qui ont été ouverts dans le bloc `try`.
- Les exceptions managées dans .NET sont implémentées par-dessus le mécanisme de gestion structurée des exceptions Win32. Pour plus d'informations, consultez les pages Gestion structurée des exceptions (C/C++) et Cours intensif sur la gestion structurée des exceptions de Win32.

Spécification du langage C#

Pour plus d'informations, consultez Exceptions dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- SystemException
- Mots clés C#
- lever
- try-catch
- try-finally
- try-catch-finally
- Exceptions

Utiliser des exceptions (Guide de programmation C#)

16/12/2020 • 6 minutes to read • Edit Online

En C#, les erreurs qui se produisent dans le programme au moment de l'exécution sont propagées à travers le programme à l'aide du mécanisme des exceptions. Les exceptions sont levées par le code qui rencontre une erreur et interceptées par le code qui peut corriger l'erreur. Les exceptions peuvent être levées par le Runtime .NET ou par le code d'un programme. Une fois qu'une exception est levée, elle se propage jusqu'en haut de la pile des appels, jusqu'à ce qu'une instruction `catch` soit trouvée pour l'exception. Les exceptions non interceptées sont gérées par un gestionnaire d'exceptions génériques fourni par le système qui affiche une boîte de dialogue.

Les exceptions sont représentées par des classes dérivées de Exception. Cette classe identifie le type d'exception et contient des propriétés comportant des détails sur l'exception. Lever une exception implique de créer une instance d'une classe dérivée d'une exception, de configurer éventuellement les propriétés de l'exception, puis de lever l'objet à l'aide du mot clé `throw`. Par exemple :

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

Après la levée d'une exception, le runtime vérifie l'instruction actuelle pour voir si elle se trouve dans un bloc `try`. Si tel est le cas, tous les blocs `catch` associés au bloc `try` font l'objet d'un contrôle permettant de déterminer s'ils peuvent intercepter l'exception. Les blocs `Catch` spécifient généralement des types d'exception ; si le type du bloc `catch` est identique à celui de l'exception ou d'une classe de base de l'exception, le bloc `catch` peut gérer la méthode. Par exemple :

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

Si l'instruction qui lève une exception ne se trouve pas dans un `try` bloc ou si le `try` bloc qui l'englobe n'a pas de `catch` bloc correspondant, le runtime vérifie la méthode d'appel pour obtenir une `try` instruction et des `catch` blocs. Le runtime continue jusqu'à la pile appelante, à la recherche d'un bloc `catch` compatible. Une fois le bloc `catch` trouvé et exécuté, le contrôle est passé à l'instruction suivante après le bloc `catch`.

Une instruction `try` peut contenir plusieurs blocs `catch`. La première `catch` instruction qui peut gérer l'exception est exécutée ; toutes les `catch` instructions suivantes, même si elles sont compatibles, sont ignorées. Les blocs catch doivent toujours être classés du plus spécifique (ou le plus dérivé) au moins spécifique. Par exemple :

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}
```

Avant que le bloc `catch` ne soit exécuté, le runtime vérifie la présence de blocs `finally`. `Finally` les blocs permettent au programmeur de nettoyer tout état ambigu qui peut être laissé à partir d'un `try` bloc abandonné, ou de libérer des ressources externes (telles que des handles graphiques, des connexions de base de données ou des flux de fichiers) sans attendre que le garbage collector dans le runtime finalise les objets. Par exemple :

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

En cas de `WriteByte()` levée d'une exception, le code du deuxième `try` bloc qui tente de rouvrir le fichier échoue si `file.Close()` n'est pas appelé, et le fichier reste verrouillé. Étant donné que les blocs `finally` sont exécutés même si une exception est levée, le bloc `finally` de l'exemple précédent autorise la fermeture correcte du fichier et permet d'éviter une erreur.

Si aucun bloc `catch` compatible n'est trouvé sur la pile des appels après la levée d'une exception, l'une des trois situations suivantes se produit :

- Si l'exception se trouve dans un finaliseur, le finaliseur est abandonné et le finaliseur de base, le cas échéant, est appelé.
- Si la pile des appels contient un constructeur statique ou un initialiseur de champ statique, une TypeInitializationException est levée, avec l'exception d'origine assignée à la propriété InnerException de la nouvelle exception.
- Si le début du thread est atteint, le thread est terminé.

Gestion des exceptions (Guide de programmation C#)

16/12/2020 • 8 minutes to read • Edit Online

Un bloc try est utilisé par les programmeurs C# pour partitionner du code susceptible d'être affecté par une exception. Des blocs catch associés sont utilisés pour gérer les exceptions générées. Un bloc finally contient du code qui est exécuté, qu'une exception soit levée ou non dans le `try` bloc, par exemple la libération de ressources qui sont allouées dans le `try` bloc. Un bloc `try` doit être associé à un ou plusieurs blocs `catch`, à un bloc `finally`, ou aux deux.

Les exemples suivants montrent une instruction `try-catch`, une instruction `try-finally` et une instruction `try-catch-finally`.

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Un bloc `try` sans bloc `catch` ou `finally` provoque une erreur du compilateur.

Blocs catch

Un bloc `catch` peut spécifier le type d'exception à intercepter. La spécification de type est appelée *filtre d'exception*. Le type d'exception doit être dérivé de Exception. En général, ne spécifiez pas Exception comme filtre d'exception, sauf si vous savez comment gérer toutes les exceptions susceptibles d'être levées dans le `try` bloc ou si vous avez

inclus une instruction throw à la fin de votre `catch` bloc.

Plusieurs `catch` blocs avec différentes classes d'exception peuvent être chaînés ensemble. Les blocs `catch` sont évalués de haut en bas dans votre code, mais un seul bloc `catch` est exécuté pour chaque exception levée. Le premier bloc `catch` qui spécifie le type exact ou une classe de base de l'exception levée est exécuté. Si aucun `catch` bloc ne spécifie une classe d'exception correspondante, un `catch` bloc qui n'a pas de type est sélectionné, s'il est présent dans l'instruction. Il est important de placer `catch` les blocs avec les classes d'exception les plus spécifiques (autrement dit, les plus dérivées).

Interceptez les exceptions lorsque les conditions suivantes sont vraies :

- Vous savez pourquoi l'exception a été levée et vous pouvez implémenter une récupération spécifique, par exemple inviter l'utilisateur à entrer un nouveau nom de fichier quand vous interceptez un objet FileNotFoundException.
- Vous pouvez créer et lever une nouvelle exception plus spécifique.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- Vous voulez traiter partiellement une exception avant de la transmettre en vue d'un traitement supplémentaire. Dans l'exemple suivant, un `catch` bloc est utilisé pour ajouter une entrée à un journal des erreurs avant de lever à nouveau l'exception.

```
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Vous pouvez également spécifier des *filtres d'exception* pour ajouter une expression booléenne à une clause catch. Ils indiquent qu'une clause catch spécifique correspond uniquement lorsque cette condition est vraie. Dans l'exemple suivant, les deux clauses catch utilisent la même classe d'exception, mais une condition supplémentaire est vérifiée pour créer un autre message d'erreur :

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Un filtre d'exception qui retourne toujours `false` peut être utilisé pour examiner toutes les exceptions, mais ne pas les traiter. Une utilisation courante consiste à enregistrer les exceptions :

```
public static void Main()
{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Exception must have been handled");
}

private static bool LogException(Exception e)
{
    Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
    Console.WriteLine($"\tMessage: {e.Message}");
    return false;
}
```

La `LogException` méthode retourne toujours `false` , aucune `catch` clause utilisant ce filtre d'exception ne correspond à. La clause catch peut être générale, l'utilisation de System.Exception et les clauses ultérieures peuvent traiter des classes d'exception plus spécifiques.

Blocs Finally

Un bloc `finally` vous permet de nettoyer les actions qui sont exécutées dans un bloc `try`. S'il est présent, le bloc `finally` s'exécute en dernier, après le bloc `try` et tout bloc `catch` mis en correspondance. Un `finally` bloc s'exécute toujours, qu'une exception soit levée ou qu'un `catch` bloc correspondant au type d'exception soit trouvé.

Le bloc `finally` peut être utilisé pour libérer des ressources telles que des flux de fichiers, des connexions de base de données et des handles graphiques, sans attendre que le récupérateur de mémoire dans le runtime finalise les objets. Pour plus d'informations, consultez l' instruction using.

Dans l'exemple suivant, le bloc `finally` est utilisé pour fermer un fichier ouvert dans le bloc `try`. Notez que l'état du handle de fichier est vérifié avant la fermeture du fichier. Si le `try` bloc ne peut pas ouvrir le fichier, le descripteur de fichier a toujours la valeur `null` et le `finally` bloc ne tente pas de le fermer. Au lieu de cela, si le fichier est ouvert avec succès dans le `try` bloc, le `finally` bloc ferme le fichier ouvert.

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

Spécification du langage C#

Pour plus d'informations, consultez Exceptions et Instruction try dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- try-catch
- try-finally
- try-catch-finally
- using, instruction

Création et levée d'exceptions (Guide de programmation C#)

16/12/2020 • 7 minutes to read • Edit Online

Les exceptions sont utilisées pour indiquer qu'une erreur s'est produite pendant l'exécution du programme. Les objets d'exception qui décrivent une erreur sont créés, puis *levés* avec le mot clé throw. Le runtime recherche ensuite le gestionnaire d'exceptions le plus compatible.

Les programmeurs doivent lever des exceptions quand une ou plusieurs des conditions suivantes sont vérifiées :

- La méthode ne peut pas terminer ses fonctionnalités définies. Par exemple, si un paramètre d'une méthode a une valeur non valide :

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be null", nameof(original));
}
```

- Un appel inapproprié à un objet est effectué en fonction de l'état de l'objet. Une tentative d'écriture dans un fichier en lecture seule en est un exemple. Dans les cas où l'état d'un objet n'autorise pas une opération, levez une instance de InvalidOperationException ou un objet basé sur une dérivation de cette classe. Le code suivant est un exemple d'une méthode qui lève un InvalidOperationException objet :

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be read-only");
        }
        // Else write data to the log and return.
    }
}
```

- Quand un argument d'une méthode provoque une exception. Dans ce cas, l'exception d'origine doit être interceptée et une instance d'ArgumentException doit être créée. L'exception d'origine doit être passée au constructeur d'ArgumentException comme paramètre InnerException :

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

Les exceptions contiennent une propriété nommée StackTrace. Cette chaîne contient le nom des méthodes sur la pile des appels actuelle, avec le nom de fichier et le numéro de ligne où l'exception a été levée pour chaque méthode. Un objet StackTrace est créé automatiquement par le CLR (Common Language Runtime) à partir du point de l'instruction `throw`, de sorte que les exceptions doivent être levées à partir du point où la trace de la pile doit commencer.

Toutes les exceptions contiennent une propriété nommée Message. Cette chaîne doit être définie pour expliquer la raison de l'exception. Les informations sensibles à la sécurité ne doivent pas être placées dans le texte du message. Outre Message, ArgumentException contient une propriété nommée ParamName dont la valeur doit être le nom de l'argument qui a provoqué la levée de l'exception. Dans un accesseur Set de propriété, ParamName doit avoir la valeur `value` .

Les méthodes publiques et protégées lèvent des exceptions chaque fois qu'elles ne peuvent pas terminer leurs fonctions prévues. La classe d'exception levée est l'exception la plus spécifique disponible qui correspond aux conditions d'erreur. Ces exceptions doivent être documentées dans le cadre de la fonctionnalité de la classe. De plus, les classes dérivées ou les mises à jour de la classe d'origine doivent conserver le même comportement afin d'assurer la compatibilité descendante.

Pratiques à éviter lors de la levée d'exceptions

La liste suivante identifie les pratiques à éviter lors de la levée d'exceptions :

- N'utilisez pas d'exceptions pour modifier le déroulement d'un programme dans le cadre d'une exécution ordinaire. Utilisez des exceptions pour signaler et gérer les conditions d'erreur.
- Les exceptions ne doivent pas être retournées en tant que valeur de retour ou paramètre au lieu d'être levées.
- Ne levez pas System.Exception ,, System.SystemException System.NullReferenceException ou System.IndexOutOfRangeException intentionnellement à partir de votre propre code source.
- Ne créez pas d'exceptions qui peuvent être levées en mode débogage, mais pas en mode release. Pour identifier des erreurs d'exécution pendant la phase de développement, utilisez plutôt Debug Assert.

Définition de classes d'exceptions

Les programmes peuvent lever une classe d'exceptions prédéfinie dans l'espace de noms System (sauf dans les endroits préalablement signalés) ou créer leurs propres classes d'exceptions en les dérivant d'Exception. Les classes dérivées doivent définir au moins trois constructeurs : un constructeur sans paramètre, un qui définit la propriété du message et un qui définit à la fois la propriété Message et la propriété InnerException. Le quatrième constructeur est utilisé pour sérialiser l'exception. Les nouvelles classes d'exception doivent être sérialisables. Par exemple :

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

Ajoutez de nouvelles propriétés à la classe d'exception lorsque les données qu'elles fournissent sont utiles pour résoudre l'exception. Si de nouvelles propriétés sont ajoutées à la classe d'exceptions dérivée, la méthode `ToString()` doit être substituée pour retourner les informations ajoutées.

Spécification du langage C#

Pour plus d'informations, consultez Exceptions et Instruction throw dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Hiérarchie des exceptions

Exceptions générées par le compilateur (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

Certaines exceptions sont levées automatiquement par le Runtime .NET lors de l'échec des opérations de base. Ces exceptions et leurs conditions d'erreur sont répertoriées dans le tableau suivant.

EXCEPTION	DESCRIPTION
ArithmeticException	Classe de base pour les exceptions qui se produisent pendant des opérations arithmétiques, telles que DivideByZeroException et OverflowException.
ArrayTypeMismatchException	Levée lorsqu'un tableau ne peut pas stocker un élément donné parce que le type réel de l'élément est incompatible avec le type réel du tableau.
DivideByZeroException	Levée lors d'une tentative de division d'une valeur intégrale par zéro.
IndexOutOfRangeException	Levée lors d'une tentative d'indexation d'un tableau à l'aide d'un index qui est inférieur à zéro ou en dehors des limites du tableau.
InvalidCastException	Levée quand une conversion explicite d'un type de base en interface ou en un type dérivé échoue au moment de l'exécution.
NullReferenceException	Levée lorsqu'une tentative est faite pour référencer un objet dont la valeur est null.
OutOfMemoryException	Levée quand une tentative d'allocation de mémoire à l'aide de l'opérateur new échoue. Cette exception indique que la mémoire disponible pour le common language runtime est épuisée.
OverflowException	Levée quand une opération dans un contexte `checked` engendre un dépassement.
StackOverflowException	Levée quand la pile d'exécution est épuisée par un trop grand nombre d'appels de méthode en attente ; cela indique généralement une récurrence très profonde ou infinie.
TypeInitializationException	Levée quand un constructeur statique lève une exception et qu'il n'existe aucune clause `catch` pour l'intercepter.

Voir aussi

- try-catch
- try-finally
- try-catch-finally

Comment gérer une exception à l'aide de try/catch (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

L'objectif d'un bloc try-catch est d'intercepter et de gérer une exception générée par du code opérationnel. Certaines exceptions peuvent être gérées dans un `catch` bloc et le problème résolu sans que l'exception soit levée à nouveau ; toutefois, plus souvent, la seule chose que vous pouvez faire est de vous assurer que l'exception appropriée est levée.

Exemple

Dans cet exemple, IndexOutOfRangeException n'est pas l'exception la plus appropriée : est ArgumentOutOfRangeException plus logique pour la méthode, car l'erreur est provoquée par l' `index` argument passé par l'appelant.

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of range.", e);
    }
}
```

Commentaires

Le code qui provoque une exception est cloisonné dans le bloc `try`. Une instruction `catch` est ajoutée juste après pour gérer `IndexOutOfRangeException`, si elle se produit. Le bloc `catch` gère `IndexOutOfRangeException` et lève l'exception plus appropriée `ArgumentOutOfRangeException` à la place. Pour fournir à l'appelant autant d'informations que possible, pensez à spécifier l'exception d'origine comme InnerException de la nouvelle exception. Étant donné que la InnerException propriété est en lecture seule, vous devez l'assigner dans le constructeur de la nouvelle exception.

Comment exécuter le code de nettoyage à l'aide de finally (Guide de programmation C#)

16/12/2020 • 2 minutes to read • Edit Online

L'objectif d'une instruction `finally` est de vérifier que le nettoyage nécessaire des objets, généralement ceux contenant des ressources externes, se produit immédiatement, même si une exception est levée. Un exemple de nettoyage de ce type est l'appel à Close sur un FileStream immédiatement après son utilisation au lieu d'attendre que l'objet soit récupéré par garbage collection par le Common Language Runtime, comme suit :

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Exemple

Pour transformer le code précédent en instruction `try-catch-finally`, le code de nettoyage est séparé du code actif comme suit.

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
    }
}
```

Comme une exception peut se produire à tout moment dans le `try` bloc avant l' `OpenWrite()` appel, ou `OpenWrite()` si l'appel lui-même peut échouer, nous n'avons pas la garantie que le fichier est ouvert lorsque nous tentons de le fermer. Le `finally` bloc ajoute une vérification pour s'assurer que l' FileStream objet ne se trouve pas `null` avant d'appeler la Close méthode. Sans la `null` vérification, le `finally` bloc pourrait lever sa propre NullReferenceException , mais la levée d'exceptions dans des `finally` blocs doit être évitée si possible.

Une connexion de base de données est également susceptible d'être fermée dans un bloc `finally`. Le nombre de connexions à un serveur de base de données étant parfois limité, vous devez fermer les connexions de base de données le plus rapidement possible. Si une exception est levée avant que vous ne puissiez fermer votre connexion, l'utilisation du `finally` bloc est préférable à l'attente de garbage collection.

Voir aussi

- using, instruction
- try-catch
- try-finally
- try-catch-finally

Comment intercepter une exception non-CLS

02/11/2020 • 2 minutes to read • Edit Online

Certains langages .NET, dont C++/CLI, permettent aux objets de lever des exceptions qui ne dérivent pas d'Exception. De telles exceptions sont appelées *exceptions non-CLS* ou *non exceptions*. Dans C#, vous ne pouvez pas lever d'exceptions non-CLS, mais vous pouvez les intercepter de deux façons :

- Dans un bloc `catch (RuntimeWrappedException e)`.

  Par défaut, un assembly Visual C# intercepte les exceptions non-CLS comme des exceptions encapsulées. Utilisez cette méthode si vous devez accéder à l'exception d'origine, qui est accessible via la propriété RuntimeWrappedException.WrappedException. La procédure située plus loin dans cette rubrique explique comment intercepter les exceptions de cette manière.

- Dans un bloc catch général (bloc catch sans type d'exception spécifié) qui est placé après tous les autres blocs `catch`.

  Utilisez cette méthode lorsque vous souhaitez effectuer une action (comme écrire dans un fichier journal) en réponse à des exceptions non-CLS, et que vous n'avez pas besoin d'accéder aux informations de l'exception. Par défaut, le common language runtime inclut dans un wrapper toutes les exceptions. Pour désactiver ce comportement, ajoutez cet attribut d'assembly à votre code, généralement dans le fichier AssemblyInfo.cs : `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

Comment intercepter une exception non-CLS

Dans un bloc `catch(RuntimeWrappedException e)`, accédez à l'exception d'origine via la propriété RuntimeWrappedException.WrappedException.

Exemple

L'exemple suivant montre comment intercepter une exception non-CLS levée à partir d'une bibliothèque de classes écrite en C++/CLI. Notez que dans cet exemple, le code client C# sait par avance que le type d'exception levé est un System.String. Vous pouvez caster la propriété RuntimeWrappedException.WrappedException en son type d'origine, tant que celui-ci est accessible à partir de votre code.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

Voir aussi

- RuntimeWrappedException
- Exceptions et gestion des exceptions

Système de fichiers et Registre (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Les articles suivants montrent comment utiliser C# et .NET pour effectuer diverses opérations de base sur les fichiers, les dossiers et le registre.

Contenu de cette section

TITRE	DESCRIPTION
Comment itérer au sein d'une arborescence de répertoires	Montre comment itérer manuellement au sein d'une arborescence de répertoires.
Comment obtenir des informations sur les fichiers, dossiers et lecteurs	Montre comment extraire des informations, par exemple l'heure de création et la taille, sur les fichiers, dossiers et lecteurs.
Comment créer un fichier ou un dossier	Montre comment créer un fichier ou un dossier.
Comment copier, supprimer et déplacer des fichiers et des dossiers (Guide de programmation C#)	Montre comment copier, supprimer et déplacer des fichiers et dossiers (C#).
Comment fournir une boîte de dialogue de progression pour les opérations de fichier	Montre comment afficher une boîte de dialogue de progression Windows standard pour certaines opérations de fichiers.
Comment écrire dans un fichier texte	Montre comment écrire dans un fichier texte.
Comment lire un fichier texte	Montre comment lire un fichier texte.
Comment lire un fichier texte ligne par ligne	Montre comment récupérer du texte dans un fichier une ligne à la fois.
Comment créer une clé dans le Registre	Montre comment écrire une clé dans le registre du système.

Sections connexes

- E/s de fichier et de flux
- Comment copier, supprimer et déplacer des fichiers et des dossiers (Guide de programmation C#)
- Guide de programmation C#
- System.IO

Comment itérer au sein d'une arborescence de répertoires (Guide de programmation C#)

02/11/2020 • 11 minutes to read • Edit Online

L'expression « itérer au sein d'une arborescence de répertoires » signifie accéder à chaque fichier dans chaque sous-répertoire imbriqué sous un dossier racine spécifié, à n'importe quelle profondeur. Vous ne devez pas nécessairement ouvrir chaque fichier. Vous pouvez simplement récupérer le nom du fichier ou du sous-répertoire sous forme de `string`, ou vous pouvez récupérer des informations supplémentaires sous la forme d'un objet System.IO.FileInfo ou System.IO.DirectoryInfo.

> **NOTE**
>
> Dans Windows, les termes « répertoire » et « dossier » sont utilisés indifféremment. La plupart de la documentation et du texte de l'interface utilisateur utilise le terme « dossier », mais les bibliothèques de classes .NET utilisent le terme « répertoire ».

Dans le cas le plus simple, où vous êtes sûr de disposer des autorisations d'accès à tous les répertoires situés sous une racine spécifiée, vous pouvez utiliser l'indicateur `System.IO.SearchOption.AllDirectories`. Cet indicateur retourne tous les sous-répertoires imbriqués qui correspondent au modèle spécifié. L'exemple suivant montre comment utiliser cet indicateur.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

La faiblesse de cette approche est que si l'un des sous-répertoires sous la racine spécifiée provoque une exception DirectoryNotFoundException ou UnauthorizedAccessException, la totalité de la méthode échoue et ne retourne pas de répertoire. Il en va de même quand vous utilisez la méthode GetFiles. Si vous devez gérer ces exceptions dans des sous-dossiers spécifiques, vous devez parcourir manuellement l'arborescence de répertoires, comme illustré dans les exemples suivants.

Quand vous parcourez manuellement une arborescence de répertoires, vous pouvez gérer d'abord les sous-répertoires (*balayage pré-ordre*), ou d'abord les fichiers (*balayage post-ordre*). Si vous effectuez un balayage pré-ordre, vous parcourez l'arborescence entière sous le dossier actif avant d'itérer au sein des fichiers qui sont directement dans le dossier lui-même. Les exemples fournis plus loin dans ce document effectuent un balayage post-ordre, mais vous pouvez les modifier facilement pour effectuer un balayage pré-ordre.

Une autre option consiste à utiliser la récurrence ou un balayage de pile. Les exemples fournis plus loin dans ce document illustrent les deux approches.

Si vous devez effectuer diverses opérations sur des fichiers et des dossiers, vous pouvez modulariser ces exemples en refactorisant l'opération dans des fonctions distinctes que vous pouvez appeler à l'aide d'un délégué unique.

NOTE

Les systèmes de fichiers NTFS peuvent contenir des *points d'analyse* sous la forme de *points de jonction*, de *liens symboliques* et de *liens physiques*. Les méthodes .NET telles que GetFiles et GetDirectories ne retournent pas de sous-répertoires sous un point d'analyse. Ce comportement vous protège contre le risque d'entrer dans une boucle infinie quand deux points d'analyse se référencent l'un l'autre. En général, lors de l'utilisation de points d'analyse vous devez faire attention à ne pas involontairement modifier ou supprimer des fichiers. Si vous avez besoin d'un contrôle précis sur les points d'analyse, utilisez du code natif ou un appel de code non managé pour appeler directement les méthodes de système de fichiers Win32 appropriées.

Exemple

L'exemple suivant montre comment parcourir une arborescence de répertoires à l'aide de la récurrence. L'approche récursive est élégante, mais susceptible de provoquer une exception de dépassement de capacité de la pile si l'arborescence de répertoires est volumineuse et profondément imbriquée.

Les exceptions particulières gérées et les actions qui sont effectuées sur chaque fichier ou dossier sont fournies à titre d'exemple uniquement. Vous devez modifier ce code pour répondre à vos besoins spécifiques. Pour plus d'informations, consultez les commentaires du code.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
```

```
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }

        if (files != null)
        {
            foreach (System.IO.FileInfo fi in files)
            {
                // In this example, we only access the existing FileInfo object. If we
                // want to open, delete or modify the file, then
                // a try-catch block is required here to handle the case
                // where the file has been deleted since the call to TraverseTree().
                Console.WriteLine(fi.FullName);
            }

            // Now find all the subdirectories under this directory.
            subDirs = root.GetDirectories();

            foreach (System.IO.DirectoryInfo dirInfo in subDirs)
            {
                // Resursive call for each subdirectory.
                WalkDirectoryTree(dirInfo);
            }
        }
    }
}
```

Exemple

L'exemple suivant montre comment itérer au sein des fichiers et des dossiers dans une arborescence de répertoires sans utiliser la récurrence. Cette technique utilise le type de collection Stack<T> générique, qui est une pile LIFO (dernier entré, premier sorti).

Les exceptions particulières gérées et les actions qui sont effectuées sur chaque fichier ou dossier sont fournies à titre d'exemple uniquement. Vous devez modifier ce code pour répondre à vos besoins spécifiques. Pour plus d'informations, consultez les commentaires du code.

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
```

```
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFound exception
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
            // choice of which exceptions to catch depends entirely on the specific task
            // you are intending to perform and also on how much you know with certainty
            // about the systems on which this code will run.
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }

            string[] files = null;
            try
            {
                files = System.IO.Directory.GetFiles(currentDir);
            }

            catch (UnauthorizedAccessException e)
            {

                Console.WriteLine(e.Message);
                continue;
            }

            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            // Perform the required action on each file here.
            // Modify this block to perform your required task.
            foreach (string file in files)
            {
                try
                {
                    // Perform whatever action is required in your scenario.
                    System.IO.FileInfo fi = new System.IO.FileInfo(file);
                    Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
                }
                catch (System.IO.FileNotFoundException e)
                {
```

```
                // If file was deleted by a separate application
                //  or thread since the call to TraverseTree()
                // then just continue.
                Console.WriteLine(e.Message);
                continue;
            }
        }

        // Push the subdirectories onto the stack for traversal.
        // This could also be done before handing the files.
        foreach (string str in subDirs)
            dirs.Push(str);
    }
  }
}
```

Cela prend généralement trop de temps de tester chaque dossier pour déterminer si votre application est autorisée à l'ouvrir. Par conséquent, l'exemple de code englobe simplement cette partie de l'opération dans un bloc `try/catch`. Vous pouvez modifier le bloc `catch` pour que, quand l'accès à un dossier vous est refusé, vous essayiez d'élever vos autorisations et d'y accéder à nouveau. En règle générale, vous devez intercepter uniquement les exceptions que vous pouvez gérer sans laisser votre application dans un état inconnu.

Si vous devez stocker le contenu d'une arborescence de répertoires en mémoire ou sur disque, la meilleure option consiste à stocker uniquement la propriété FullName (de type `string`) pour chaque fichier. Vous pouvez ensuite utiliser cette chaîne pour créer un objet FileInfo ou DirectoryInfo, si nécessaire, ou ouvrir les fichiers qui nécessitent un traitement supplémentaire.

Programmation fiable

Tout code d'itération de fichiers fiable doit tenir compte de nombreuses complexités du système de fichiers. Pour plus d'informations sur le système de fichiers Windows, consultez Vue d'ensemble de NTFS.

Voir aussi

- System.IO
- LINQ et répertoires de fichiers
- Système de fichiers et Registre (Guide de programmation C#)

Comment obtenir des informations sur les fichiers, dossiers et lecteurs (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Dans .NET, vous pouvez accéder aux informations de système de fichiers à l'aide des classes suivantes :

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

Les classes FileInfo et DirectoryInfo représentent un fichier ou un répertoire. Elles contiennent des propriétés qui exposent une grande partie des attributs de fichiers pris en charge par le système de fichiers NTFS. Elles contiennent également des méthodes pour ouvrir, fermer, déplacer et supprimer des fichiers et dossiers. Vous pouvez créer des instances de ces classes en passant une chaîne qui représente le nom du fichier, dossier ou lecteur dans le constructeur :

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

Vous pouvez aussi obtenir les noms des fichiers, dossiers ou lecteurs en appelant DirectoryInfo.GetDirectories, DirectoryInfo.GetFiles et DriveInfo.RootDirectory.

Les classes System.IO.Directory et System.IO.File fournissent des méthodes statiques pour récupérer des informations sur les répertoires et les fichiers.

Exemple

L'exemple suivant illustre différentes manières d'accéder aux informations sur les fichiers et dossiers.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }
```

```
        // Get the subdirectories directly that is under the root.
        // See "How to: Iterate Through a Directory Tree" for an example of how to
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.*");

        foreach (System.IO.DirectoryInfo d in dirInfos)
        {
            Console.WriteLine(d.Name);
        }

        // The Directory and File classes provide several static methods
        // for accessing files and directories.

        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileInfo objects.
        // Use this method when storage space is an issue, and when you might
        // hold on to the file name reference for a while before you try to access
        // the file.
        string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

        foreach (string s in files)
        {
            // Create the FileInfo object only when needed to ensure
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if the
        // directory cannot be found.
        if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
        {
            System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
        }

        System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

        currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Programmation fiable

Quand vous traitez des chaînes de chemin spécifiées par l'utilisateur, vous devez également gérer les exceptions dans les cas suivants :

- Le nom de fichier est mal formé. Par exemple, il contient des caractères non valides ou uniquement des espaces blancs.
- Le nom de fichier est null.
- Le nom de fichier est plus long que la longueur maximale définie par le système.
- Le nom de fichier contient un signe deux-points (:).

Si l'application ne dispose pas des autorisations suffisantes pour lire le fichier spécifié, la méthode `Exists` retourne `false` indépendamment de l'existence d'un chemin. La méthode ne lève pas d'exception.

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)

Comment créer un fichier ou un dossier (Guide de programmation C#)

02/11/2020 • 5 minutes to read • Edit Online

Vous pouvez par programmationcréer un dossier sur votre ordinateur, créer un sous-dossier, créer un fichier dans le sous-dossier et écrire des données dans le fichier.

Exemple

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
```

```
        }
        else
        {
            Console.WriteLine("File \"{0}\" already exists.", fileName);
            return;
        }

        // Read and display the data from your file.
        try
        {
            byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
            foreach (byte b in readBuffer)
            {
                Console.Write(b + " ");
            }
            Console.WriteLine();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
    // Sample output:

    // Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

    //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    //30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
    // 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
    //3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}
```

Si le dossier existe déjà, CreateDirectory est sans effet et aucune exception n'est levée. Toutefois, File.Create remplace un fichier existant par un nouveau fichier. L'exemple utilise une instruction `if` - `else` pour éviter qu'un fichier existant soit pas remplacé.

En apportant les modifications suivantes dans l'exemple, vous pouvez spécifier des résultats différents si un fichier avec un nom spécifique existe déjà. Si un tel fichier n'existe pas, le code en crée un. Si un tel fichier existe, le code ajoute des données à ce fichier.

- Spécifiez un nom de fichier non aléatoire.

```
// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";
```

- Remplacez l'instruction `if` - `else` par l'instruction `using` dans le code suivant.

```
using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}
```

Exécutez l'exemple plusieurs fois pour vérifier que les données sont ajoutées au fichier à chaque fois.

Pour découvrir d'autres valeurs `FileMode` que vous pouvez essayer, consultez FileMode.

Les conditions ci-dessous peuvent générer une exception.

- Le nom du dossier est mal formé. Par exemple, il contient des caractères non conformes ou uniquement des espaces blancs (classe ArgumentException). Utilisez la classe Path pour créer des noms de chemin valides.
- Le dossier parent du dossier à créer est en lecture seule (classe IOException).
- Le nom du dossier est `null` (classe ArgumentNullException).
- Le nom du dossier est trop long (classe PathTooLongException).
- Le nom du dossier est uniquement un signe deux-points, « : » (classe PathTooLongException).

Sécurité .NET

Une instance de la classe SecurityException peut être levée dans les situations de confiance partielle.

Si vous n'êtes pas autorisé à créer le dossier, l'exemple lève une instance de la UnauthorizedAccessException classe.

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)

Comment copier, supprimer et déplacer des fichiers et des dossiers (Guide de programmation C#)

02/11/2020 • 4 minutes to read • Edit Online

Les exemples suivants montrent comment copier, déplacer et supprimer des fichiers et dossiers de manière synchrone à l'aide des classes System.IO.File, System.IO.Directory, System.IO.FileInfo et System.IO.DirectoryInfo à partir de l'espace de noms System.IO. Ces exemples ne fournissent pas de barre de progression ou autre interface utilisateur. Si vous souhaitez fournir une boîte de dialogue de progression standard, consultez Guide pratique pour fournir une boîte de dialogue de progression pour les opérations sur les fichiers.

Utilisez System.IO.FileSystemWatcher pour fournir des événements qui vous permettent de calculer la progression quand vous travaillez sur plusieurs fichiers. Une autre approche consiste à utiliser l'appel de code non managé pour appeler les méthodes pertinentes relatives aux fichiers dans le shell Windows. Pour plus d'informations sur la façon d'effectuer ces opérations sur les fichiers de façon asynchrone, consultez E/S de fichiers asynchrones.

Exemple

L'exemple suivant montre comment copier des fichiers et des répertoires.

```
// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath =  @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Exemple

L'exemple suivant montre comment déplacer des fichiers et des répertoires.

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}
```

Exemple

L'exemple suivant montre comment supprimer des fichiers et des répertoires.

```
// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
```

```
        {
            Console.WriteLine(e.Message);
        }
        // Delete a directory and all subdirectories with Directory static method...
        if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
        {
            try
            {
                System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
            }

            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }
        }

        // ...or with DirectoryInfo instance method.
        System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
        // Delete this dir and all subdirs.
        try
        {
            di.Delete(true);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)
- Comment fournir une boîte de dialogue de progression pour les opérations de fichier
- E/s de fichier et de flux
- Tâches d'e/s courantes

Guide pratique pour fournir une boîte de dialogue de progression pour les opérations sur les fichiers (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Vous pouvez fournir une boîte de dialogue standard qui montre la progression des opérations sur les fichiers dans Windows si vous utilisez la méthode CopyFile(String, String, UIOption) de l'espace de noms Microsoft.VisualBasic.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Pour ajouter une référence dans Visual Studio

1. Dans la barre de menus, choisissez **Projet**, **Ajouter une référence**.

   La boîte de dialogue **Gestionnaire de références** s'affiche.

2. Dans la zone **Assemblys**, choisissez **Framework** s'il n'est pas déjà sélectionné.

3. Dans la liste des noms, cochez la case **Microsoft.VisualBasic**, puis choisissez le bouton **OK** pour fermer la boîte de dialogue.

Exemple

Le code suivant copie le répertoire spécifié par `sourcePath` dans le répertoire spécifié par `destinationPath`. Ce code fournit également une boîte de dialogue standard qui affiche une estimation du temps restant avant la fin de l'opération.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

Voir aussi

- Système de fichiers et Registre (Guide de programmation C#)

Comment écrire dans un fichier texte (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Ces exemples montrent différentes manières d'écrire du texte dans un fichier. Les deux premiers exemples utilisent des méthodes pratiques statiques au niveau de la classe System.IO.File pour écrire chaque élément de tout `IEnumerable<string>` et une chaîne dans un fichier texte. L'exemple 3 indique comment ajouter du texte à un fichier quand vous devez traiter chaque ligne individuellement pendant que vous écrivez dans le fichier. Les exemples 1 à 3 remplacent tout le contenu existant dans le fichier, alors que l'exemple 4 montre comment ajouter du texte à un fichier existant.

Ces exemples écrivent tous des littéraux de chaîne dans des fichiers. Pour mettre en forme le texte écrit dans un fichier, utilisez la méthode Format ou la fonctionnalité d'interpolation de chaîne C#.

Exemple

```
class WriteTextFile
{
    static void Main()
    {

        // These examples assume a "C:\Users\Public\TestFolder" folder on your machine.
        // You can modify the path if necessary.

        // Example #1: Write an array of strings to a file.
        // Create a string array that consists of three lines.
        string[] lines = { "First line", "Second line", "Third line" };
        // WriteAllLines creates a file, writes a collection of strings to the file,
        // and then closes the file.  You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllLines(@"C:\Users\Public\TestFolder\WriteLines.txt", lines);

        // Example #2: Write one string to a text file.
        string text = "A class is the most powerful data type in C#. Like a structure, " +
                      "a class defines the data and behavior of the data type. ";
        // WriteAllText creates a file, writes the specified string to the file,
        // and then closes the file.    You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllText(@"C:\Users\Public\TestFolder\WriteText.txt", text);

        // Example #3: Write only some strings in an array to a file.
        // The using statement automatically flushes AND CLOSES the stream and calls
        // IDisposable.Dispose on the stream object.
        // NOTE: do not use FileStream for text files because it writes bytes, but StreamWriter
        // encodes the output as text.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt"))
        {
            foreach (string line in lines)
            {
                // If the line doesn't contain the word 'Second', write the line to the file.
                if (!line.Contains("Second"))
                {
                    file.WriteLine(line);
                }
            }
        }

        // Example #4: Append new text to an existing file.
        // The using statement automatically flushes AND CLOSES the stream and calls
```

```
        // IDisposable.Dispose on the stream object.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt", true))
        {
            file.WriteLine("Fourth line");
        }
    }
}
 //Output (to WriteLines.txt):
 //   First line
 //   Second line
 //   Third line

 //Output (to WriteText.txt):
 //   A class is the most powerful data type in C#. Like a structure, a class defines the data and behavior of
the data type.

 //Output to WriteLines2.txt after Example #3:
 //   First line
 //   Third line

 //Output to WriteLines2.txt after Example #4:
 //   First line
 //   Third line
 //   Fourth line
```

Programmation fiable

Les conditions ci-dessous peuvent générer une exception.

- Le fichier existe déjà et est en lecture seule.
- Le nom du chemin d'accès peut s'avérer trop long.
- Le disque est peut-être plein.

Voir aussi

- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)
- Exemple : Enregistrer une collection sur un emplacement de stockage d'application

Comment lire un fichier texte (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple lit le contenu d'un fichier texte à l'aide des méthodes statiques ReadAllText et ReadAllLines de la classe System.IO.File.

Pour obtenir un exemple qui utilise StreamReader , consultez Comment lire un fichier texte ligne par ligne.

> **NOTE**
>
> Les fichiers utilisés dans cet exemple sont créés dans la rubrique Comment écrire dans un fichier texte.

Exemple

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compilation du code

Copiez le code et collez-le dans une application console C#.

Si vous n'utilisez pas les fichiers texte à partir de la procédure d'écriture dans un fichier texte, remplacez l'argument par `ReadAllText` et `ReadAllLines` par le chemin d'accès et le nom de fichier appropriés sur votre ordinateur.

Programmation fiable

Les conditions ci-dessous peuvent générer une exception.

- Le fichier n'existe pas ou ne se trouve pas à l'emplacement spécifié. Vérifiez le chemin et l'orthographe du nom de fichier.

Sécurité .NET

Ne comptez pas sur le nom d'un fichier pour en déduire le contenu. Par exemple, le fichier `myFile.cs` peut ne pas être un fichier source C#.

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)

Guide pratique pour lire un fichier texte ligne par ligne (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

Cet exemple lit le contenu d'une chaîne d'un fichier texte, ligne par ligne, à l'aide de la méthode `ReadLine` de la classe `StreamReader`. Chaque ligne de texte est stockée dans la chaîne `line` et s'affiche à l'écran.

Exemple

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compilation du code

Copiez le code et collez-le dans la méthode `Main` d'une application console.

Remplacez `"c:\test.txt"` par le nom du fichier.

Programmation fiable

Les conditions ci-dessous peuvent générer une exception.

- Le fichier n'existe pas.

Sécurité .NET

Ne vous basez pas sur le nom d'un fichier pour en déterminer le contenu. Par exemple, le fichier `myFile.cs` peut ne pas être un fichier source C#.

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)

Comment créer une clé dans le registre (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Cet exemple ajoute la paire de valeurs « Name » et « Isabella » dans le Registre de l'utilisateur actuel, sous la clé « Names ».

Exemple

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Compilation du code

- Copiez le code et collez-le dans la méthode `Main` d'une application console.
- Remplacez le paramètre `Names` par le nom d'une clé qui existe directement sous le nœud HKEY_CURRENT_USER du Registre.
- Remplacez le paramètre `Name` par le nom d'une valeur qui existe directement sous le nœud Names.

Programmation fiable

Examinez la structure du Registre pour rechercher un emplacement approprié pour votre clé. Par exemple, vous souhaiterez peut-être ouvrir la clé Software de l'utilisateur actuel et créer une clé avec le nom de votre société. Ensuite, ajoutez les valeurs de Registre à la clé de votre société.

Les conditions ci-dessous peuvent générer une exception :

- Le nom de la clé est null.
- L'utilisateur ne dispose pas des autorisations nécessaires pour créer des clés de Registre.
- Le nom de la clé dépasse la limite de 255 caractères.
- La clé est fermée.
- La clé de Registre est en lecture seule.

Sécurité .NET

Il est plus sûr d'écrire des données dans le dossier utilisateur (`Microsoft.Win32.Registry.CurrentUser`) que sur l'ordinateur local (`Microsoft.Win32.Registry.LocalMachine`).

Quand vous créez une valeur de Registre, vous devez déterminer ce qu'il faut faire si cette valeur existe déjà. Il est possible qu'un autre processus, éventuellement malveillant, ait déjà créé la valeur et y ait accès. Quand vous placez des données dans la valeur de Registre, ces données sont accessibles à l'autre processus. Pour éviter ce problème, utilisez la méthode `Overload:Microsoft.Win32.RegistryKey.GetValue` . Elle retourne null si la clé n'existe pas encore.

Il est dangereux de stocker des données confidentielles, telles que des mots de passe, dans le Registre sous forme

de texte brut, même si la clé de Registre est protégée par des listes de contrôle d'accès.

Voir aussi

- System.IO
- Guide de programmation C#
- Système de fichiers et Registre (Guide de programmation C#)
- Read, write and delete from the registry with C# (Lire, écrire et supprimer du Registre avec C#)

Interopérabilité (Guide de programmation C#)

02/11/2020 • 2 minutes to read • Edit Online

L'interopérabilité vous permet de préserver et de tirer parti d'investissements existants en code non managé. Le code qui s'exécute sous le contrôle du common language runtime (CLR) est appelé *code managé*, et le code qui s'exécute en dehors du CLR est appelé *code non managé*. COM, COM+, les composants C++, les composants ActiveX et l'API Microsoft Windows sont des exemples de code non managé.

.NET permet l'interopérabilité avec du code non managé via des services d'appel de code non managé, l' System.Runtime.InteropServices espace de noms, l'interopérabilité C++ et l'interopérabilité COM (COM Interop).

Dans cette section

Vue d'ensemble de l'interopérabilité
Décrit les méthodes qui permettent l'interopérabilité entre le code managé et le code non managé en C#.

Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#
Décrit les fonctionnalités introduites dans Visual C# pour faciliter la programmation Office.

Comment utiliser des propriétés indexées dans la programmation COM Interop
Explique comment utiliser des propriétés indexées pour accéder aux propriétés COM qui ont des paramètres.

Comment utiliser l'appel de code non managé pour lire un fichier WAV
Explique comment utiliser des services d'appel de code non managé pour lire un fichier son .wav sur le système d'exploitation Windows.

Procédure pas à pas : programmation Office
Montre comment créer un classeur Excel et un document Word qui contient un lien vers le classeur.

Exemple de classe COM
Montre comment exposer une classe C# en tant qu'objet COM.

Spécification du langage C#

Pour plus d'informations, consultez Concepts de base dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Marshal.ReleaseComObject
- Guide de programmation C#
- Interopération avec du code non managé
- Procédure pas à pas : programmation Office

Vue d'ensemble de l'interopérabilité (Guide de programmation C#)

02/11/2020 • 6 minutes to read • Edit Online

Cette rubrique décrit les méthodes qui permettent une interopérabilité entre le code managé C# et le code non managé.

Appel de plateforme

L'*appel de code non managé* est un service qui permet au code managé d'appeler des fonctions non managées implémentées dans des bibliothèques de liens dynamiques (DLL), telles que celles de l'API Windows Microsoft. Il localise et appelle une fonction exportée, puis marshale ses arguments (entiers, chaînes, tableaux, structures, etc) au-delà des limites d'interopérabilité, selon les besoins.

Pour plus d'informations, consultez consommation de fonctions DLL non managées et utilisation de l'appel de code non managé pour lire un fichier WAV.

> **NOTE**
>
> Le common language runtime (CLR) gère l'accès aux ressources système. Le fait d'appeler du code non managé extérieur au CLR contourne ce mécanisme, ce qui présente un risque de sécurité. Par exemple, du code non managé peut appeler directement des ressources dans du code non managé en contournant les mécanismes de sécurité CLR. Pour plus d'informations, consultez Sécurité dans .NET.

Interopérabilité C++

Vous pouvez utiliser l'interopérabilité C++, également connue sous le nom de Just Works (IJW), pour encapsuler une classe C++ native afin qu'elle puisse être utilisée par du code créé en C# ou un autre langage .NET. Pour ce faire, écrivez du code C++ pour encapsuler un composant DLL ou COM natif. Contrairement à d'autres langages .NET, Visual C++ a une prise en charge de l'interopérabilité qui permet de placer du code managé et non managé dans la même application et même dans le même fichier. Vous pouvez ensuite générer le code C++ à l'aide du commutateur de compilateur **/clr** pour créer un assembly managé. Enfin, ajoutez une référence à l'assembly dans votre projet C# et utilisez les objets encapsulés comme vous le feriez pour les autres classes managées.

Exposer des composants COM au langage C#

Vous pouvez utiliser un composant COM d'un projet C#. Les étapes générales sont les suivantes :

1. Recherchez le composant COM à utiliser et enregistrez-le. Utilisez regsvr32.exe pour inscrire ou désinscrire une DLL COM.

2. Ajoutez au projet une référence au composant COM ou à la bibliothèque de types.

   Quand vous ajoutez la référence, Visual Studio utilise le Tlbimp.exe (importateur de bibliothèques de types), qui prend une bibliothèque de types comme entrée pour générer un assembly .NET Interop. L'assembly, également appelé wrapper RCW, contient des classes et des interfaces managées qui encapsulent les classes et les interfaces COM qui se trouvent dans la bibliothèque de types. Visual Studio ajoute au projet une référence à l'assembly généré.

3. Créez une instance d'une classe qui est définie dans le wrapper RCW. Une instance de l'objet COM est ainsi

créée.

4. Utilisez l'objet comme vous utiliseriez tout autre objet managé. Lorsque l'objet est récupéré par le garbage collection, l'instance de l'objet COM est également libérée de la mémoire.

Pour plus d'informations, consultez Exposition de composants COM au .NET Framework.

Exposition du langage C# à COM

Les clients COM peuvent utiliser des types C# qui ont été correctement exposés. Les étapes de base pour exposer des types C# sont les suivantes :

1. Ajoutez des attributs d'interopérabilité dans le projet C#.

   Vous pouvez rendre visible un assembly COM en modifiant les propriétés de projet Visual C#. Pour plus d'informations, consultez Informations de l'assembly, boîte de dialogue.

2. Générez une bibliothèque de types COM et inscrivez-la pour l'utilisation de COM.

   Vous pouvez modifier les propriétés de projet Visual C# pour inscrire automatiquement l'assembly C# pour COM Interop. Visual Studio utilise le fichier Regasm.exe (outil d'inscription de l'assembly) avec le commutateur de ligne de commande `/tlb`, qui prend comme entrée un assembly managé pour générer une bibliothèque de types. Cette bibliothèque de types décrit les types `public` de l'assembly et ajoute les entrées du Registre pour que les clients COM puissent créer des classes managées.

Pour plus d'informations, consultez Exposition de composants .NET Framework à COM et Exemple de classe COM.

Voir aussi

- Améliorer les performances d'interopérabilité
- Introduction à l'interopérabilité entre COM et .NET
- Introduction à COM Interop en Visual Basic
- Marshaling entre du code managé et non managé
- Interopération avec du code non managé
- Guide de programmation C#

Comment accéder aux objets Office Interop (Guide de programmation C#)

02/11/2020 • 20 minutes to read • Edit Online

C# possède des fonctionnalités qui simplifient l'accès aux objets d'API Office. Les nouvelles fonctionnalités incluent les arguments nommés et les arguments facultatifs, un nouveau type appelé `dynamic` et la possibilité de passer des arguments aux paramètres de référence dans les méthodes COM comme s'il s'agissait de paramètres de valeur.

Dans cette rubrique, vous allez utiliser les nouvelles fonctionnalités pour écrire le code qui crée et affiche une feuille de calcul Microsoft Office Excel. Vous écrirez ensuite le code pour ajouter un document Office Word qui contient une icône liée à la feuille de calcul Excel.

Pour effectuer cette procédure pas à pas, Microsoft Office Excel 2007 et Microsoft Office Word 2007, ou versions ultérieures, doivent être installés sur votre ordinateur.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Pour créer une application console

1. Démarrez Visual Studio.
2. Dans le menu **Fichier** , pointez sur **Nouveau**, puis cliquez sur **Projet**. La boîte de dialogue **Nouveau projet** apparaît.
3. Dans le volet **Modèles installés**, développez **Visual C#**, puis cliquez sur **Windows**.
4. Vérifiez en haut de la boîte de dialogue **Nouveau projet** que **.NET Framework 4** (ou version ultérieure) est sélectionné comme version cible de .NET Framework.
5. Dans le volet **Modèles**, cliquez sur **Application console**.
6. Attribuez un nom à votre projet dans le champ **Nom**.
7. Cliquez sur **OK**.

   Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Pour ajouter des références

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis cliquez sur **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche.
2. Dans la page **Assemblys**, sélectionnez **Microsoft.Office.Interop.Word** dans la liste **Nom du composant**, puis maintenez la touche CTRL enfoncée et sélectionnez **Microsoft.Office.Interop.Excel**. Si vous ne voyez pas les assemblys, vous devrez peut-être vous assurer qu'ils sont installés et affichés. Consultez Comment : installer les assemblys PIA (Primary Interop Assembly) d'Office.

3. Cliquez sur **OK**.

Pour ajouter les directives using nécessaires

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier *Program.cs*, puis cliquez sur **Afficher le code**.

2. Ajoutez les `using` directives suivantes en haut du fichier de code :

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Pour créer une liste de comptes bancaires

1. Copiez-collez la définition de classe suivante dans **Program.cs**, sous la classe `Program`.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Ajoutez le code suivant à la méthode `Main` pour créer une liste `bankAccounts` contenant deux comptes.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Pour déclarer une méthode qui exporte les informations de compte vers Excel

1. Ajoutez la méthode suivante à la classe `Program` pour définir une feuille de calcul Excel.

   La méthode Add a un paramètre facultatif pour spécifier un modèle particulier. Les paramètres optionnels, introduits dans C# 4, vous permettent d'omettre l'argument du paramètre, si vous souhaitez utiliser la valeur par défaut de ce dernier. Dans la mesure où aucun argument n'est envoyé dans le code suivant, `Add` utilise le modèle par défaut et crée un classeur. L'instruction équivalente dans les versions antérieures de C# nécessite un argument d'espace réservé : `ExcelApp.Workbooks.Add(Type.Missing)`.

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a praticular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. À la fin de `DisplayInExcel`, ajoutez le code suivant. Le code insère les valeurs dans les deux premières colonnes de la première ligne de la feuille de calcul.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. À la fin de `DisplayInExcel`, ajoutez le code suivant. La boucle `foreach` place les informations de la liste des comptes dans les deux premières colonnes des lignes successives de la feuille de calcul.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. À la fin de `DisplayInExcel`, ajoutez le code suivant pour ajuster les largeurs de colonne au contenu.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

Les versions antérieures de C# nécessitent un cast explicite pour ces opérations, car `ExcelApp.Columns[1]` retourne `Object`, et `AutoFit` est une méthode Range d'Excel. Les lignes suivantes illustrent le cast.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# 4 et versions ultérieures convertit automatiquement le retourné `Object` en `dynamic` si l'assembly est référencé par l'option du compilateur -Link ou, de manière équivalente, si la propriété Excel **incorporer les types Interop** a la valeur true. La valeur par défaut de cette propriété est true.

Pour exécuter le projet

1. À la fin de `Main`, ajoutez la ligne suivante.

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. Appuyez sur CTRL+F5.

   Une feuille de calcul Excel s'affiche avec les données des deux comptes.

Pour ajouter un document Word

1. Pour illustrer d'autres façons grâce auxquelles C# 4, ainsi que les versions ultérieures, améliore la programmation Office, le code suivant ouvre une application Word et crée une icône liée à la feuille de calcul Excel.

   Collez la méthode `CreateIconInWordDoc`, fournie ultérieurement dans cette étape, dans la classe `Program`. `CreateIconInWordDoc` utilise des arguments nommés et facultatifs pour réduire la complexité des appels de méthode à Add et PasteSpecial. Ces appels incorporent deux autres nouvelles fonctionnalités introduites dans C# 4, qui simplifient les appels aux méthodes COM comportant des paramètres de référence. Premièrement, vous pouvez envoyer des arguments aux paramètres de référence comme s'il s'agissait de paramètres de valeur. Autrement dit, vous pouvez envoyer les valeurs directement, sans créer une variable pour chaque paramètre de référence. Le compilateur génère des variables temporaires pour stocker les valeurs d'argument et les ignore lors du retour de l'appel. Ensuite, vous pouvez omettre le mot clé `ref` dans la liste d'arguments.

   La méthode `Add` comporte quatre paramètres de référence, tous facultatifs. Dans C# 4.0 ou versions ultérieures, vous pouvez omettre les arguments de tout ou partie des paramètres si vous voulez utiliser leurs valeurs par défaut. Dans C# 3.0 et versions antérieures, un argument doit être fourni pour chaque paramètre et l'argument doit être une variable, car les paramètres sont des paramètres de référence.

   La méthode `PasteSpecial` insère le contenu du Presse-papiers. La méthode comporte sept paramètres de référence, tous facultatifs. Le code suivant spécifie des arguments pour deux d'entre eux : `Link` pour créer un lien vers la source du contenu du Presse-papiers et `DisplayAsIcon` pour afficher le lien sous forme d'icône. Dans C# 4.0 et versions ultérieures, vous pouvez utiliser des arguments nommés pour ces deux-là et omettre les autres. Bien qu'il s'agisse de paramètres de référence, vous n'avez pas à utiliser le mot clé `ref` ou à créer des variables à envoyer tant qu'arguments. Vous pouvez envoyer les valeurs directement. Dans C# 3.0 et versions antérieures, vous devez fournir un argument de variable pour chaque paramètre de référence.

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

   Dans C# 3.0 ou dans les versions antérieures du langage, le code plus complexe suivant est requis.

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. À la fin de `Main`, ajoutez l'instruction suivante.

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. À la fin de `DisplayInExcel`, ajoutez l'instruction suivante. La méthode `Copy` ajoute la feuille de calcul au Presse-papiers.

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. Appuyez sur CTRL+F5.

   Un document Word s'affiche avec une icône. Double-cliquez sur l'icône pour afficher la feuille de calcul au premier plan.

Pour définir la propriété Incorporer les types interop

1. D'autres améliorations sont possibles lorsque vous appelez un type COM qui ne nécessite pas un assembly PIA (Primary Interop Assembly) au moment de l'exécution. La suppression de la dépendance vis-à-vis des assemblys PIA aboutit à l'indépendance des versions et facilite le déploiement. Pour plus d'informations sur les avantages de la programmation sans assemblys PIA, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

En outre, la programmation est plus facile, car les types qui sont requis et retournés par les méthodes COM peuvent être représentés par l'utilisation du type `dynamic` à la place d'`Object`. Les variables de type `dynamic` ne sont pas évaluées avant l'exécution, ce qui élimine la nécessité d'un cast explicite. Pour plus d'informations, consultez Utilisation du type dynamic.

Dans C# 4, l'incorporation des informations de type au lieu de l'utilisation des assemblys PIA est le comportement par défaut. En raison de ce comportement par défaut, plusieurs des exemples précédents sont simplifiés, car un cast explicite n'est pas requis. Par exemple, la déclaration de `worksheet` dans `DisplayInExcel` est écrite sous la forme `Excel._Worksheet workSheet = excelApp.ActiveSheet` plutôt que sous la forme `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. Les appels à `AutoFit` dans la même méthode nécessiteraient également un cast explicite sans la valeur par défaut, parce que `ExcelApp.Columns[1]` retourne un `Object` et que `AutoFit` est une méthode Excel. Le code suivant illustre le cast.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. Pour modifier la valeur par défaut et utiliser les assemblys PIA au lieu d'incorporer les informations de type, développez le nœud **Références** dans **l'Explorateur de solutions**, puis sélectionnez **Microsoft.Office.Interop.Excel** ou **Microsoft.Office.Interop.Word**.

3. Si la fenêtre **Propriétés** n'apparaît pas, appuyez sur **F4**.

4. Recherchez **Incorporer les types interop** dans la liste des propriétés et attribuez-lui la valeur **False**. De manière équivalente, vous pouvez compiler à l'aide de l'option de compilateur -Reference à la place de -Link à une invite de commandes.

Pour ajouter une mise en forme supplémentaire au tableau

1. Remplacez les deux appels à `AutoFit` de `DisplayInExcel` avec l'instruction suivante.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

La méthode AutoFormat a sept paramètres de valeur, tous facultatifs. Les arguments nommés et les arguments facultatifs vous permettent de fournir des arguments pour tout ou partie des paramètres, voire pour aucun. Dans l'instruction précédente, un argument est fourni pour un seul des paramètres, `Format`. Comme `Format` est le premier paramètre dans la liste des paramètres, vous n'avez pas à fournir le nom du paramètre. Cependant, l'instruction peut être plus facile à comprendre si le nom du paramètre est inclus, comme illustré dans le code suivant.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Appuyez sur CTRL+F5 pour afficher le résultat. D'autres formats sont listés dans l'énumération XlRangeAutoFormat.

3. Comparez l'instruction de l'étape 1 au code suivant, qui indique les arguments requis dans C# 3.0 et versions antérieures.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Exemple

L'exemple de code suivant illustre l'exemple complet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
```

```
            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy method has one
            // optional parameter for specifying a destination. Because no argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to create
            // variables to send in as arguments. You can send the values directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

Voir aussi

- Type.Missing
- dynamic
- Utilisation du type dynamic
- Arguments nommés et facultatifs
- Comment : utiliser des arguments nommés et facultatifs dans la programmation Office

Comment utiliser des propriétés indexées en COM Interop programmation (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Les *propriétés indexées* améliorent la façon dont les propriétés COM avec des paramètres sont consommées dans la programmation C#. Les propriétés indexées fonctionnent avec d'autres fonctionnalités dans Visual C#, comme les arguments nommés et facultatifs, un nouveau type (dynamique) et les informations de type incorporées, pour améliorer la programmation Microsoft Office.

Dans les versions antérieures de C#, les méthodes sont accessibles comme des propriétés uniquement si la méthode `get` n'a aucun paramètre et que la méthode `set` a un seul et unique paramètre de valeur. Toutefois, toutes les propriétés COM ne respectent pas ces restrictions. Par exemple, la propriété Range[Object, Object] d'Excel a un accesseur `get`, qui nécessite un paramètre pour le nom de la plage. Dans le passé, parce que vous ne pouviez pas accéder à la propriété `Range` directement, vous deviez utiliser la méthode `get_Range` à la place, comme indiqué dans l'exemple suivant.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Les propriétés indexées vous permettent d'écrire ce qui suit à la place :

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

L'exemple précédent utilise également la fonctionnalité des arguments facultatifs, qui vous permet d'omettre `Type.Missing`.

De même, pour définir la valeur de la propriété `Value` d'un objet Range dans C# 3.0 et les versions antérieures, deux arguments sont nécessaires. L'un fournit un argument pour un paramètre facultatif qui spécifie le type de la valeur de la plage. L'autre fournit la valeur de la propriété `Value`. Les exemples suivants illustrent ces techniques. Les deux définissent la valeur de la cellule A1 sur `Name`.

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Les propriétés indexées vous permettent d'écrire le code suivant à la place.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

Vous ne pouvez pas créer vos propres propriétés indexées. La fonctionnalité prend uniquement en charge la consommation de propriétés indexées existantes.

Exemple

L'exemple de code suivant illustre l'exemple complet. Pour plus d'informations sur la configuration d'un projet qui accède à l'API Office, consultez Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#.

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

Voir aussi

- Arguments nommés et facultatifs
- dynamic
- Utilisation du type dynamic
- Comment : utiliser des arguments nommés et facultatifs dans la programmation Office
- Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#
- Procédure pas à pas : programmation Office

Comment utiliser l'appel de code non managé pour lire un fichier WAV (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

L'exemple de code C# suivant montre comment utiliser les services d'appel de code non managé pour lire un fichier audio WAV sur le système d'exploitation Windows.

Exemple

Cet exemple de code utilise DllImportAttribute pour importer le point d'entrée de la méthode `PlaySound` de `winmm.dll` sous la forme `Form1 PlaySound()`. L'exemple utilise un formulaire Windows Form simple avec un bouton. En cliquant sur le bouton, vous ouvrez une boîte de dialogue OpenFileDialog Windows standard qui permet d'ouvrir un fichier à lire. Lorsqu'un fichier Wave est sélectionné, il est lu à l'aide `PlaySound()` de la méthode de la bibliothèque de *winmm.dll*. Pour plus d'informations sur cette méthode, consultez Utilisation de la fonction PlaySound avec des fichiers audio Waveform. Recherchez et sélectionnez un fichier avec une extension .wav, puis cliquez sur **Ouvrir** pour lire le fichier audio à l'aide de l'appel de code non managé. Une zone de texte affiche le chemin complet du fichier sélectionné.

La boîte de dialogue **Ouvrir les fichiers** est filtrée pour afficher uniquement les fichiers avec une extension .wav au moyen des paramètres de filtre :

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  // Constructor.
        {
            InitializeComponent();
        }

        [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet = CharSet.Unicode,
ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            var dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if (dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Including this empty method in the sample because in the IDE,
            // when users click on the form, generates code that looks for a default method
            // with this name. We add it here to prevent confusion for those using the samples.
        }
    }
}
```

Compilation du code

1. Créez un nouveau projet d'application de Windows Forms C# dans Visual Studio et nommez-le **winsound**.

2. Copiez le code ci-dessus et collez-le sur le contenu du fichier *Form1.cs* .

3. Copiez le code suivant et collez-le dans le fichier *Form1.Designer.cs* , dans la `InitializeComponent()`

méthode, après tout code existant.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compilez, puis exécutez le code.

Voir aussi

- Guide de programmation C#
- Vue d'ensemble de l'interopérabilité
- Présentation détaillée de l’appel de code non managé
- Marshaling de données à l’aide de l’appel de code managé

Procédure pas à pas : Programmation Office (C# et Visual Basic)

02/11/2020 • 21 minutes to read • Edit Online

Visual Studio offre des fonctionnalités dans C# et Visual Basic qui améliorent la programmation Microsoft Office. Les fonctionnalités utiles de C# incluent des arguments nommés et facultatifs ainsi que des valeurs de retour de type `dynamic`. Dans la programmation COM, vous pouvez omettre le mot clé `ref` et accéder aux propriétés indexées. Les fonctionnalités de Visual Basic incluent les propriétés implémentées automatiquement, les instructions dans les expressions lambda et les initialiseurs de collection.

Les deux langages autorisent l'incorporation d'informations de type, qui permet de déployer des assemblys interagissant avec les composants COM sans déployer les assemblys PIA (Primary Interop Assemblies) sur l'ordinateur de l'utilisateur. Pour plus d'informations, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

Cette procédure pas à pas illustre ces fonctionnalités dans le contexte de la programmation Office, mais beaucoup d'entre elles sont aussi utiles en programmation générale. Dans la procédure pas à pas, vous allez utiliser une application de complément Excel pour créer un classeur Excel. Vous créerez ensuite un document Word contenant un lien vers le classeur. Enfin, vous apprendrez à activer et désactiver la dépendance d'assembly PIA.

Prérequis

Pour effectuer cette procédure pas à pas, Microsoft Office Excel et Microsoft Office Word doivent être installés sur votre ordinateur.

> **NOTE**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Pour configurer une application de type complément Excel

1. Démarrez Visual Studio.
2. Dans le menu **Fichier** , pointez sur **Nouveau**, puis cliquez sur **Projet**.
3. Dans le volet **Modèles installés**, développez **Visual Basic** ou **Visual C#**, **Office**, puis cliquez sur l'année de version du produit Office.
4. Dans le volet **modèles** , cliquez **sur <version> complément Excel**.
5. Regardez en haut du volet **Modèles** pour vérifier que **.NET Framework 4**, ou version ultérieure, apparaît dans la zone **Framework cible**.
6. Tapez un nom pour votre projet dans la zone **Nom**, si vous le souhaitez.
7. Cliquez sur **OK**.
8. Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Pour ajouter des références

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis cliquez sur **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche.

2. Sous l'onglet **Assemblys**, sélectionnez **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (pour plus d'informations sur les numéros de version des produits Office, consultez Versions Microsoft),dans la liste **Nom du composant**, puis maintenez la touche CTRL enfoncée et sélectionnez **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. Si les assemblys n'apparaissent pas, vous devez vérifier qu'ils sont installés et s'affichent (consultez Guide pratique pour installer les assemblys PIA (Primary Interop Assembly) d'Office).

3. Cliquez sur **OK**.

Pour ajouter les instructions Imports ou les directives using nécessaires

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier **ThisAddIn.vb** ou **ThisAddIn.cs**, puis cliquez sur **Afficher le code**.

2. Ajoutez les instructions `Imports` suivantes (Visual Basic) ou les directives `using` (C#) en haut du fichier de code, si elles ne sont pas déjà présentes.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

Pour créer une liste de comptes bancaires

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, cliquez sur **Ajouter**, puis sur **Classe**. Nommez la classe Account.vb si vous utilisez Visual Basic ou Account.cs si vous utilisez C#. Cliquez sur **Add**.

2. Remplacez la définition de la classe `Account` par le code suivant : Les définitions de classe utilisent les *propriétés implémentées automatiquement*. Pour plus d'informations, consultez Propriétés implémentées automatiquement.

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. Pour créer une `bankAccounts` liste qui contient deux comptes, ajoutez le code suivant à la `ThisAddIn_Startup` méthode dans *ThisAddIn. vb* ou *ThisAddIn.cs*. Les déclarations de liste utilisent les *initialiseurs de collection*. Pour plus d'informations, consultez Initialiseurs de collection.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
                          .ID = 345,
                          .Balance = 541.27
                      },
    New Account With {
                          .ID = 123,
                          .Balance = -127.44
                      }
    }
```

Pour exporter des données vers Excel

1. Dans le même fichier, ajoutez la méthode suivante à la classe `ThisAddIn`. La méthode configure un classeur Excel, vers lequel elle exporte les données.

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
               ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

Deux nouvelles fonctionnalités C# sont utilisées dans cette méthode. Ces deux fonctionnalités existent déjà en Visual Basic.

- La méthode Add possède un *paramètre facultatif* pour spécifier un modèle particulier. Les paramètres optionnels, introduits dans C# 4, vous permettent d'omettre l'argument du paramètre, si vous souhaitez utiliser la valeur par défaut de ce dernier. Dans la mesure où aucun argument n'est envoyé dans l'exemple précédent, `Add` utilise le modèle par défaut et crée un classeur. L'instruction équivalente dans les versions antérieures de C# nécessite un argument d'espace réservé : `excelApp.Workbooks.Add(Type.Missing)`.

  Pour plus d'informations, consultez arguments nommés et facultatifs.

- Les propriétés `Range` et `Offset` de l'objet Range utilisent la fonctionnalité des *propriétés indexées*. Cette fonctionnalité vous permet de consommer ces propriétés à partir des types COM à l'aide de la syntaxe C# standard suivante. Les propriétés indexées vous permettent également d'utiliser la propriété `Value` de l'objet `Range`, ce qui rend superflue l'utilisation de la propriété `Value2`. La propriété `Value` est indexée, mais l'index est optionnel. Les arguments facultatifs et les propriétés indexées fonctionnent ensemble dans l'exemple suivant.

  ```
  // Visual C# 2010 provides indexed properties for COM programming.
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  Dans les versions antérieures du langage, la syntaxe spéciale suivante est requise.

  ```
  // In Visual C# 2008, you cannot access the Range, Offset, and Value
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

  Vous ne pouvez pas créer vos propres propriétés indexées. La fonctionnalité prend uniquement en charge la consommation de propriétés indexées existantes.

  Pour plus d'informations, consultez Guide pratique pour utiliser des propriétés indexées dans la programmation de COM Interop.

2. À la fin de `DisplayInExcel`, ajoutez le code suivant pour ajuster les largeurs de colonne au contenu.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

Ces ajouts illustrent une autre fonctionnalité de C# : le traitement des valeurs `Object` retournées par les hôtes COM tels qu'Office comme si leur type était dynamic. Cela se produit automatiquement lorsque l'option **incorporer les types d'interopérabilité** est définie sur sa valeur par défaut, `True` ou, de manière équivalente, lorsque l'assembly est référencé par l'option du compilateur -Link . Le type `dynamic` permet la liaison tardive, déjà disponible dans Visual Basic et évite le cast explicite requis dans C# 3.0 et versions antérieures du langage.

Par exemple, `excelApp.Columns[1]` retourne un `Object` et `AutoFit` est une méthode Range Excel. Sans `dynamic`, vous devez effectuer un cast de l'objet retourné par `excelApp.Columns[1]` sous la forme d'une instance de `Range` avant d'appeler la méthode `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

Pour plus d'informations sur l'incorporation des types d'interopérabilité, consultez les procédures « Pour rechercher la référence d'assembly PIA » et « Pour restaurer la dépendance d'assembly PIA » plus loin dans cette rubrique. Pour plus d'informations sur `dynamic`, consultez dynamic ou Utilisation du type dynamic.

Pour appeler DisplayInExcel

1. Ajoutez le code suivant à la fin de la méthode `ThisAddIn_StartUp`. L'appel à `DisplayInExcel` contient deux arguments. Le premier argument est le nom de la liste des comptes à traiter. Le deuxième argument est une expression lambda multiligne qui définit comment les données doivent être traitées. Les valeurs `ID` et `balance` de chaque compte s'affichent dans des cellules adjacentes et la ligne s'affiche en rouge si le solde est inférieur à zéro. Pour plus d'informations, consultez expressions lambda.

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
       Sub(account, cell)
           ' This multiline lambda expression sets custom
           ' processing rules for the bankAccounts.
           cell.Value = account.ID
           cell.Offset(0, 1).Value = account.Balance

           If account.Balance < 0 Then
               cell.Interior.Color = RGB(255, 0, 0)
               cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
           End If
       End Sub)
```

2. Pour exécuter le programme, appuyez sur F5. Une feuille de calcul Excel s'affiche avec les données des comptes.

Pour ajouter un document Word

1. Ajoutez le code suivant à la fin de la méthode `ThisAddIn_StartUp` pour créer un document Word qui contient un lien vers le classeur Excel.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

Ce code illustre plusieurs fonctionnalités nouvelles en C# : la possibilité d'omettre le mot clé `ref` dans la programmation COM, les arguments nommés et les arguments facultatifs. Ces fonctionnalités existent déjà en Visual Basic. La méthode PasteSpecial comporte sept paramètres, tous définis en tant que paramètres de référence facultatifs. Les arguments nommés et les arguments facultatifs vous permettent de désigner les paramètres auxquels vous souhaitez accéder par leur nom et d'envoyer des arguments à ces seuls paramètres. Dans cet exemple, les arguments sont envoyés pour indiquer qu'un lien vers le classeur dans le Presse-papiers doit être créé (paramètre `Link`) et que ce lien doit être affiché dans le document Word sous forme d'une icône (paramètre `DisplayAsIcon`). Visual C# vous permet également d'omettre le mot clé `ref` pour ces arguments.

Pour exécuter l'application

1. Appuyez sur F5 pour exécuter l'application. Excel démarre et affiche un tableau qui contient les informations des deux comptes de `bankAccounts`. Puis, un document Word apparaît qui contient un lien vers le tableau Excel.

Pour nettoyer le projet terminé

1. Dans Visual Studio, cliquez sur **Nettoyer la solution** dans le menu **Générer**. Sinon, le complément s'exécutera chaque fois que vous ouvrirez Excel sur votre ordinateur.

Pour rechercher la référence d'assembly PIA

1. Exécutez de nouveau l'application, mais ne cliquez pas sur **Nettoyer la solution**.

2. Sélectionnez le bouton **Démarrer**. Recherchez **Microsoft Visual Studio <version> ** et ouvrez une invite de commandes développeur.

3. Tapez `ildasm` dans la fenêtre Invite de commandes développeur pour Visual Studio, puis appuyez sur Entrée. La fenêtre IL DASM s'affiche.

4. Dans le menu **Fichier** de la fenêtre IL DASM, sélectionnez **Fichier** > **Ouvrir**. Double-cliquez sur **Visual <version> Studio **, puis double-cliquez sur **projets**. Ouvrez le dossier de votre projet et, dans le dossier bin/Debug, recherchez *nom de votre projet*.dll. Double-cliquez sur *nom de votre projet*.dll. Une nouvelle fenêtre affiche les attributs de votre projet, en plus des références à d'autres modules et assemblys. Remarquez que les espaces de noms `Microsoft.Office.Interop.Excel` et `Microsoft.Office.Interop.Word` sont inclus dans l'assembly. Par défaut, dans Visual Studio, le compilateur importe les types dont vous avez besoin à partir d'un assembly PIA référencé dans votre assembly.

   Pour plus d'informations, consultez Guide pratique pour afficher le contenu d'un assembly.

5. Double-cliquez sur l'icône **MANIFESTE**. Une fenêtre affiche la liste des assemblys contenant les éléments référencés par le projet. `Microsoft.Office.Interop.Excel` et `Microsoft.Office.Interop.Word` ne sont pas inclus dans la liste. Étant donné que les types dont votre projet a besoin ont été importés dans votre assembly, les références à un assembly PIA ne sont pas requis. Le déploiement s'en trouve facilité. Les assemblys PIA ne doivent pas être présents sur l'ordinateur de l'utilisateur ; comme une application ne nécessite pas le déploiement d'une version spécifique d'un assembly PIA, les applications peuvent être conçues pour fonctionner avec plusieurs versions d'Office, sous réserve que les API nécessaires existent dans toutes les versions.

   Le déploiement d'assemblys PIA n'étant plus nécessaire, vous pouvez créer une application dans les scénarios avancés qui fonctionne avec plusieurs versions d'Office, y compris les versions antérieures. Cependant, cela ne fonctionne que si votre code n'utilise pas d'API qui ne sont pas disponibles dans la version d'Office que vous utilisez. Comme il n'est pas toujours évident de savoir si une API particulière était disponible dans une version antérieure, l'utilisation de versions antérieures d'Office n'est pas recommandée.

   > **NOTE**
   > Jusqu'à Office 2003, Office ne publiait pas les assemblys PIA. Par conséquent, le seul moyen de générer un assembly d'interopérabilité pour Office 2002 ou versions antérieures consiste à importer la référence COM.

6. Fermez la fenêtre de manifeste et la fenêtre d'assembly.

Pour restaurer la dépendance d'assembly PIA

1. Dans l'**Explorateur de solutions**, cliquez sur le bouton **Afficher tous les fichiers**. Développez le dossier **Références** et sélectionnez **Microsoft.Office.Interop.Excel**. Appuyez sur F4 pour afficher la fenêtre **Propriétés**.

2. Dans la fenêtre **Propriétés**, remplacez la valeur **True** de la propriété **Incorporer les types d'interopérabilité** par **False**.

3. Répétez les étapes 1 et 2 de cette procédure pour `Microsoft.Office.Interop.Word`.

4. Dans C#, commentez les deux appels à `Autofit` à la fin de la méthode `DisplayInExcel`.

5. Appuyez sur F5 pour vérifier que le projet continue de s'exécuter correctement.

6. Répétez les étapes 1 à 3 de la procédure précédente pour ouvrir la fenêtre d'assembly. Notez que `Microsoft.Office.Interop.Word` et `Microsoft.Office.Interop.Excel` ne sont plus dans la liste des assemblys incorporés.

7. Double-cliquez sur l'icône **MANIFESTE** et faites défiler la liste des assemblys référencés. `Microsoft.Office.Interop.Word` et `Microsoft.Office.Interop.Excel` figurent tous deux dans la liste. Comme

l'application référence les assemblys PIA Excel et Word, et que la propriété **Incorporer les types d'interopérabilité** a la valeur **False**, les deux assemblys doivent exister sur l'ordinateur de l'utilisateur final.

8. Dans Visual Studio, cliquez sur **Nettoyer la solution** dans le menu **Générer** pour nettoyer le projet achevé.

Voir aussi

- Propriétés implémentées automatiquement (Visual Basic)
- Propriétés implémentées automatiquement (C#)
- Initialiseurs de collection
- Initialiseurs d'objets et de collections
- Paramètres facultatifs
- Passage des arguments par position et par nom
- Arguments nommés et facultatifs
- Liaison précoce et tardive
- dynamic
- Utilisation du type dynamic
- Expressions lambda (Visual Basic)
- Expressions lambda (C#)
- Comment utiliser des propriétés indexées dans la programmation COM Interop
- Procédure pas à pas : incorporation d'informations de type provenant d'assemblys Microsoft Office dans Visual Studio
- Procédure pas à pas : incorporation de types provenant d'assemblys managés
- Procédure pas à pas : création de votre premier complément VSTO pour Excel
- COM Interop
- Interopérabilité

Exemple de classe COM (Guide de programmation C#)

02/11/2020 • 3 minutes to read • Edit Online

Voici un exemple de classe pouvant être exposée en tant qu'objet COM. Après avoir placé ce code dans un fichier .cs et l'avoir ajouté à votre projet, affectez la valeur **True** à la propriété **Inscrire pour COM Interop**. Pour plus d'informations, consultez Guide pratique pour inscrire un composant pour COM Interop.

L'exposition d'objets Visual C# à COM nécessite de déclarer une interface de classe, une interface d'événements si nécessaire, et la classe proprement dite. Les membres de classe doivent suivre ces règles pour être visibles par COM :

- La classe doit être publique.
- Les propriétés, méthodes et événements doivent être publics.
- Les propriétés et méthodes doivent être déclarées dans l'interface de classe.
- Les événements doivent être déclarés dans l'interface d'événement.

Les autres membres publics de la classe qui ne sont pas déclarés dans ces interfaces ne seront pas visibles par COM, mais ils seront visibles par d'autres objets .NET.

Pour exposer des propriétés et des méthodes à COM, vous devez les déclarer sur l'interface de classe, les marquer avec un attribut `DispId` et les implémenter dans la classe. L'ordre dans lequel les membres sont déclarés dans l'interface est l'ordre utilisé pour la vtable COM.

Pour exposer des événements à partir de votre classe, vous devez les déclarer sur l'interface d'événement et les marquer avec un attribut `DispId`. La classe ne doit pas implémenter cette interface.

La classe implémente l'interface de classe ; elle peut implémenter plusieurs interfaces, mais la première implémentation sera l'interface de classe par défaut. Implémentez ici les méthodes et propriétés exposées à COM. Elles doivent être marquées comme publiques et doivent correspondre aux déclarations dans l'interface de classe. Vous devez aussi déclarer ici les événements déclenchés par la classe. Ils doivent être marqués comme publics et doivent correspondre aux déclarations dans l'interface d'événement.

Exemple

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

Voir aussi

- Guide de programmation C#
- Interopérabilité
- Générer, page du Concepteur de projets (C#)

Informations de référence sur C#

02/11/2020 • 6 minutes to read • Edit Online

Cette section fournit des ressources de référence sur les mots clés, les opérateurs, les caractères spéciaux, les directives de préprocesseur, les options du compilateur, et les erreurs et avertissements du compilateur du langage C#.

Contenu de cette section

Mots clés C#

Fournit des liens vers des informations sur la syntaxe et les mots clés C#.

Opérateurs C#

Fournit des liens vers des informations sur la syntaxe et les opérateurs C#.

Caractères spéciaux C#

Fournit des liens vers des informations sur des caractères spéciaux contextuels en C# et leur utilisation.

Directives de préprocesseur C#

Fournit des liens vers des informations sur les commandes de compilateur permettant d'incorporer du code source en C#.

Options du compilateur C#

Inclut des informations sur les options du compilateur et comment les utiliser.

Erreurs du compilateur C#

Inclut des extraits de code qui montrent la cause et la correction des erreurs du compilateur et des avertissements C#.

Spécification du langage C#

Spécification du langage C# 6.0. Il s'agit d'une ébauche de proposition pour le langage C# 6.0. Ce document sera affiné par le biais du Comité des normes ECMA C#. La version 5.0 a été publiée en décembre 2017 dans le document Standard ECMA-334 5th Edition.

Les fonctionnalités ayant été implémentées dans les versions C# ultérieures à 6.0 sont représentées dans des propositions de spécification du langage. Ces documents décrivent les delta de la spécification du langage afin d'ajouter ces nouvelles fonctionnalités. Il s'agit d'un formulaire de proposition. Ces spécifications seront affinées et soumises au Comité des normes ECMA pour une révision et une incorporation formelles dans une future version de la norme C#.

Propositions de spécification C# 7,0

De nouvelles fonctionnalités ont été implémentées dans C# 7.0 : critères spéciaux, fonctions locales, déclarations de variables out, expressions throw, littéraux binaires et séparateurs numériques notamment. Ce dossier contient les spécifications de chacune de ces fonctionnalités.

Propositions de spécification C# 7,1

De nouvelles fonctionnalités ont été ajoutées à C# 7.1. D'abord, vous pouvez écrire une méthode `Main` qui renvoie `Task` ou `Task<int>`. Ceci vous permet d'ajouter le modificateur `async` à `Main`. L'expression `default` peut être utilisée sans type dans des emplacements où

le type peut être déduit. De plus, les noms de membres de tuple peuvent être déduits. Enfin, les critères spéciaux peuvent être utilisés avec des génériques.

Propositions de spécification C# 7,2

C# 7.2 apporte un certain nombre de petites fonctionnalités. Vous pouvez envoyer des arguments par référence en lecture seule à l'aide du mot clé `in`. Il existe un certain nombre de modifications de bas niveau pour soutenir la sécurité au moment de la compilation pour `Span` et d'autres types associés. Vous pouvez utiliser les arguments nommés où des arguments ultérieurs seront positionnels dans certains cas. Le modificateur d'accès `private protected` vous permet de spécifier que les appelants sont limités à des types dérivés implémentés dans le même assembly. L'opérateur `?:` peut se servir d'une référence pour une variable. Vous pouvez aussi formater des nombres binaires et hexadécimaux à l'aide d'un séparateur numérique de début.

Propositions de spécification C# 7,3

C#7.3 est une nouvelle version qui comprend plusieurs petites mises à jour. Vous pouvez utiliser de nouvelles contraintes sur les paramètres de type générique. D'autres modifications facilitent l'utilisation des `fixed` champs, y compris l'utilisation des `stackalloc` allocations. Les variables locales déclarées avec le `ref` mot clé peuvent être réassignées pour faire référence au nouveau stockage. Vous pouvez placer des attributs sur les propriétés implémentées automatiquement qui ciblent le champ de stockage généré par le compilateur. Les variables d'expression peuvent être utilisées dans les initialiseurs. Les tuples peuvent être comparés pour savoir s'ils sont égaux (ou inégaux). Il y a également eu des améliorations pour la résolution de surcharge.

Propositions de spécification C# 8,0

C# 8,0 est disponible avec .NET Core 3,0. Les fonctionnalités incluent les types de référence Nullable, les critères spéciaux récursifs, les méthodes d'interface par défaut, les flux asynchrones, les plages et les index, les modèles basés sur et les déclarations, l'affectation de fusion Null et les membres d'instance en lecture seule.

Sections connexes

Utilisation de l'environnement de développement Visual Studio pour C#

Fournit des liens vers des des rubriques de concepts et de tâches qui décrivent l'IDE et l'éditeur.

Guide de programmation C#

Inclut des informations sur l'utilisation du langage de programmation C#.

Gestion des versions du langage C#

16/12/2020 • 9 minutes to read • Edit Online

Le compilateur C# le plus récent détermine une version du langage par défaut en fonction du ou des frameworks cibles de votre projet. Visual Studio ne fournit pas d'interface utilisateur pour modifier la valeur, mais vous pouvez la modifier en modifiant le fichier *csproj* . Le choix de la valeur par défaut garantit que vous utilisez la dernière version de langage compatible avec votre Framework cible. Vous bénéficiez d'un accès aux fonctionnalités de langage les plus récentes compatibles avec la cible de votre projet. Ce choix par défaut garantit également que vous n'utilisez pas un langage qui requiert des types ou un comportement d'exécution non disponible dans votre version cible de .NET Framework. Si vous choisissez une version de langue plus récente que la valeur par défaut, il est difficile de diagnostiquer les erreurs de compilation et d'exécution.

Les règles de cet article s'appliquent au compilateur fourni avec Visual Studio 2019 ou le kit de développement logiciel (SDK) .NET. Les compilateurs C# qui sont installés en même temps que Visual Studio 2017 ou les versions antérieures du kit SDK .NET Core ciblent C# 7.0 par défaut.

C# 8,0 est pris en charge uniquement sur .NET Core 3. x et les versions plus récentes. La plupart des fonctionnalités les plus récentes requièrent des fonctionnalités de bibliothèque et d'exécution introduites dans .NET Core 3. x :

- L'implémentation de l' interface par défaut nécessite de nouvelles fonctionnalités dans le CLR .net Core 3,0.
- Les flux asynchrones requièrent les nouveaux types System.IAsyncDisposable , System.Collections.Generic.IAsyncEnumerable<T> et System.Collections.Generic.IAsyncEnumerator<T> .
- Les index et les plages nécessitent les nouveaux types System.Index et System.Range .
- Les types de référence Nullable utilisent plusieurs attributs pour fournir de meilleurs avertissements. Ces attributs ont été ajoutés dans .NET Core 3,0. D'autres frameworks cibles n'ont pas été annotés avec l'un de ces attributs. Cela signifie que les avertissements Nullable peuvent ne pas refléter avec précision les problèmes potentiels.

C# 9,0 est pris en charge uniquement sur .NET 5 et les versions plus récentes.

Valeurs par défaut

Le compilateur détermine une valeur par défaut en fonction de ces règles :

VERSION CIBLE DE .NET FRAMEWORK	VERSION	VERSION DU LANGAGE C# PAR DÉFAUT
.NET	5	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3

VERSION CIBLE DE .NET FRAMEWORK	VERSION	VERSION DU LANGAGE C# PAR DÉFAUT
.NET Framework	all	C# 7.3

Lorsqu'il s'agit d'une préversion cible pour laquelle il existe une préversion du langage correspondante, c'est cette dernière qui est utilisée. Vous utilisez les fonctionnalités les plus récentes avec cette préversion dans n'importe quel environnement, sans affecter les projets qui ciblent une version .NET Core publiée.

> **IMPORTANT**
> Visual Studio 2017 a ajouté une `<LangVersion>latest</LangVersion>` entrée aux fichiers projet qu'il a créés. Cela signifiait *C# 7,0* quand il a été ajouté. Toutefois, une fois que vous avez mis à niveau vers Visual Studio 2019, cela signifie la dernière version publiée, quelle que soit la version cible de .NET Framework. Ces projets remplacent désormais le comportement par défaut. Vous devez modifier le fichier projet et supprimer ce nœud. Votre projet utilisera ensuite la version du compilateur recommandée pour votre version cible de .NET Framework.

Remplacer les valeurs par défaut

Si vous devez spécifier votre version C# explicitement, vous pouvez le faire de plusieurs façons :

- Modifiez manuellement votre fichier projet.
- Définissez la version du langage pour plusieurs projets d'un sous-répertoire.
- Configurez l' `-langversion` option du compilateur.

> **TIP**
> Pour connaître la version linguistique que vous utilisez actuellement, mettez- `#error version` la (respecte la casse) dans votre code. Ainsi, le compilateur produit un diagnostic, CS8304, avec un message contenant la version du compilateur utilisée et la version de langage actuellement sélectionnée.

Modifier le fichier projet

Vous pouvez définir la version du langage dans votre fichier projet. Par exemple, si vous souhaitez accéder explicitement aux fonctionnalités d'évaluation, ajoutez un élément comme suit :

```
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

La valeur `preview` utilise la dernière préversion disponible du langage C# que prend en charge votre compilateur.

Configurer plusieurs projets

Pour configurer plusieurs projets, vous pouvez créer un fichier **Directory. Build. props** qui contient l' `<LangVersion>` élément. En règle générale, vous faites cela dans votre répertoire de solution. Ajoutez le code suivant à un fichier **Directory. Build. props** dans le répertoire de votre solution :

```
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

Les builds dans tous les sous-répertoires du répertoire contenant ce fichier utiliseront la version d'évaluation en C#. Pour plus d'informations, consultez l'article sur Personnaliser votre build.

Informations de référence sur la version du langage C#

Le tableau suivant montre toutes les versions actuelles du langage C#. Votre compilateur peut ne pas nécessairement comprendre chaque valeur si elle est plus ancienne. Si vous installez le dernier Kit de développement logiciel (SDK) .NET, vous avez accès à tout ce qui est listé.

VALEUR	SIGNIFICATION
`preview`	Le compilateur accepte toute la syntaxe de langage valide de la dernière préversion.
`latest`	Le compilateur accepte la syntaxe de la dernière version publiée du compilateur (versions mineures incluses).
`latestMajor` (`default`)	Le compilateur accepte la syntaxe de la dernière version principale publiée du compilateur.
`9.0`	Le compilateur accepte uniquement la syntaxe qui est incluse dans C# 9,0 ou une partie antérieure.
`8.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 8.0 ou une version antérieure.
`7.3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.3 ou une version antérieure.
`7.2`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.2 ou une version antérieure.
`7.1`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.1 ou une version antérieure.
`7`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.0 ou une version antérieure.
`6`	Le compilateur accepte uniquement la syntaxe incluse dans C# 6.0 ou une version antérieure.
`5`	Le compilateur accepte uniquement la syntaxe incluse dans C# 5.0 ou une version antérieure.
`4`	Le compilateur accepte uniquement la syntaxe incluse dans C# 4.0 ou une version antérieure.
`3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 3.0 ou une version antérieure.
`ISO-2` (ou `2`)	Le compilateur accepte uniquement la syntaxe qui est incluse dans la norme ISO/IEC 23270:2006 C# (2,0).
`ISO-1` (ou `1`)	Le compilateur accepte uniquement la syntaxe qui est incluse dans la norme ISO/IEC 23270:2003 C# (1.0/1.2).

TIP

Ouvrez le invite de commandes développeur pour Visual Studio, puis exécutez la commande suivante pour afficher la liste des versions de langue disponibles sur votre ordinateur.

```
csc -langversion:?
```

En cas de question de l'option de compilation-langversion comme ceci, imprimera un résultat similaire à ce qui suit :

```
Supported language versions:
default
1
2
3
4
5
6
7.0
7.1
7.2
7.3
8.0
9.0 (default)
latestmajor
preview
latest
```

Types valeur (référence C#)

16/12/2020 • 6 minutes to read • Edit Online

Les types *valeur* et les types référence sont les deux principales catégories de types C#. Une variable d'un type valeur contient une instance du type. Cela diffère d'une variable d'un type référence, qui contient une référence à une instance du type. Par défaut, lors de l' assignation, en passant un argument à une méthode et en retournant un résultat de méthode, les valeurs des variables sont copiées. Dans le cas des variables de type valeur, les instances de type correspondantes sont copiées. L'exemple suivant illustre ce comportement :

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Comme le montre l'exemple précédent, les opérations sur une variable de type valeur affectent uniquement cette instance du type valeur, stockée dans la variable.

Si un type valeur contient un membre de données d'un type référence, seule la référence à l'instance du type référence est copiée lorsqu'une instance de type valeur est copiée. La copie et l'instance de type valeur d'origine ont accès à la même instance de type référence. L'exemple suivant illustre ce comportement :

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

NOTE

Pour que votre code soit moins sujet aux erreurs et plus robuste, définissez et utilisez des types de valeurs immuables. Cet article utilise des types de valeur mutable uniquement à des fins de démonstration.

Genres de types valeur et contraintes de type

Un type valeur peut être l'un des deux types suivants :

- type structure, qui encapsule les données et les fonctionnalités associées
- type énumération, qui est défini par un ensemble de constantes nommées et qui représente un choix ou une combinaison de choix

Un type valeur Nullable `T?` représente toutes les valeurs de son type valeur sous-jacent `T` et une valeur null supplémentaire. Vous ne pouvez pas assigner `null` à une variable d'un type valeur, sauf s'il s'agit d'un type valeur Nullable.

Vous pouvez utiliser la `struct` contrainte pour spécifier qu'un paramètre de type est un type valeur qui n'autorise pas les valeurs NULL. Les types de structure et d'énumération satisfont à la `struct` contrainte. À compter de C# 7,3, vous pouvez utiliser `System.Enum` dans une contrainte de classe de base (appelée contrainte enum) pour spécifier qu'un paramètre de type est un type énumération.

Types valeur intégrés

C# fournit les types valeur intégrés suivants, également appelés *types simples*:

- Types numériques intégraux
- Types numériques à virgule flottante
- booléen qui représente une valeur booléenne
- char qui représente un caractère Unicode UTF-16

Tous les types simples sont des types structure et diffèrent des autres types de structure en ce qu'ils autorisent certaines opérations supplémentaires :

- Vous pouvez utiliser des littéraux pour fournir une valeur d'un type simple. Par exemple, `'A'` est un littéral de type `char`, tandis que `2001` est un littéral de type `int`.
- Vous pouvez déclarer des constantes des types simples avec le mot clé const. Il n'est pas possible d'avoir des constantes d'autres types de structure.
- Les expressions constantes, dont les opérandes sont toutes des constantes des types simples, sont évaluées au moment de la compilation.

À compter de C# 7,0, C# prend en charge les tuples de valeur. Un tuple de valeur est un type valeur, mais pas un type simple.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types de valeur
- Types simples
- Variables

Voir aussi

- Informations de référence sur C#
- System.ValueType
- Types référence

Types numériques intégraux (référence C#)

21/04/2020 • 6 minutes to read • Edit Online

Les *types numériques intégrals* représentent les nombres d'intégraires. Tous les types numériques intégral sont des types de valeur. Ils sont également des types simples et peuvent être parascés avec des littérals. Tous les types numériques intégral prennent en charge l'arithmétique, la logique bitwise, la comparaison,et les opérateurs d'égalité.

Caractéristiques des types intégraux

C# prend en charge les types intégraux prédéfinis suivants :

C# TYPE/MOT CLÉ	PLAGE	TAILLE	TYPE .NET
`sbyte`	-128 à 127	Entier 8 bits signé	System.SByte
`byte`	0 à 255	Entier 8 bits non signé	System.Byte
`short`	-32 768 à 32 767	Entier 16 bits signé	System.Int16
`ushort`	0 à 65 535	Entier 16 bits non signé	System.UInt16
`int`	-2 147 483 648 à 2 147 483 647	Entier 32 bits signé	System.Int32
`uint`	de 0 à 4 294 967 295	Entier 32 bits non signé	System.UInt32
`long`	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	Entier 64 bits signé	System.Int64
`ulong`	de 0 à 18 446 744 073 709 551 615	Entier 64 bits non signé	System.UInt64

Dans le tableau précédent, chaque mot clé de type C# de la colonne la plus à gauche est un alias pour le type .NET correspondant. Ils sont interchangeables. Par exemple, les déclarations suivantes déclarent des variables du même type :

```
int a = 123;
System.Int32 b = 123;
```

La valeur par défaut pour chaque type intégral est zéro, `0`. Chacun des types intégraux a les constantes `MinValue` et `MaxValue` qui fournissent la valeur minimale et maximale de ce type.

Utilisez la structure System.Numerics.BigInteger pour représenter un entier signé sans limites supérieures ou inférieures.

Littéraux d'entier

Les littérals integer peuvent être

- *décimale*: sans préfixe
- *hexadecimal*: `0x` avec `0X` le préfixe ou le préfixe
- *binaire*: `0b` avec `0B` le ou le préfixe (disponible en C 7.0 et plus tard)

Le code suivant montre un exemple de chacun :

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

L'exemple précédent montre `_` également l'utilisation d'un *séparateur à chiffres*, qui est pris en charge à partir de C 7.0. Vous pouvez utiliser le séparateur de chiffre avec toutes sortes de littéral numériques.

Le type d'un littéral integer est déterminé par son suffixe comme suit:

- Si le littéral n'a pas de suffixe, son type est le `int` `uint` premier `long` `ulong` des types suivants dans lesquels sa valeur peut être représentée: , , , .
- Si le littéral est `U` `u` suffixe par ou, son type est le premier des `uint` `ulong` types suivants dans lesquels sa valeur peut être représentée: , .
- Si le littéral est `L` `l` suffixe par ou, son type est le premier des `long` `ulong` types suivants dans lesquels sa valeur peut être représentée: , .

  > **NOTE**
  > Vous pouvez utiliser la `l` lettre minuscule comme suffixe. Cependant, cela génère un avertissement compilateur parce que la lettre `l` peut être confondue avec le chiffre `1`. Utiliser `L` pour plus de clarté.

- Si le littéral est `UL` `Ul` suffixe par `Lu`, `lu`, `ulong`, `uL` `ul`, `LU`, , `lU`, , ou , son type est .

Si la valeur représentée par un littéral entier dépasse UInt64.MaxValue, une erreur de compilation CS1021 se produit.

Si le type déterminé d'un `int` littéral integer est et la valeur représentée par le littéral est `sbyte` `byte` dans `short` `ushort` la gamme du type de destination, la valeur peut être implicitement convertie en , , , , `uint`, ou: `ulong`

```
byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a 'byte'
```

Comme l'exemple précédent le montre, si la valeur du littéral n'est pas dans la plage du type de destination, une erreur de compilateur CS0031 se produit.

Vous pouvez également utiliser un plâtre pour convertir la valeur représentée par un

littéral integer au type autre que le type déterminé de littéral:

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversions

Vous pouvez convertir n'importe quel type numérique intégral à n'importe quel autre type numérique intégral. Si le type de destination peut stocker toutes les valeurs du type source, la conversion est implicite. Sinon, vous devez utiliser une expression de fonte pour effectuer une conversion explicite. Pour plus d'informations, voir conversions numériques intégrées.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types intégraux
- Littéraux d'entier

Voir aussi

- Référence C#
- Types de valeur
- Types virgule flottante
- Chaînes de format numériques standard
- Valeurs numériques dans .NET

Types numériques à virgule flottante (Référence C#)

02/11/2020 • 6 minutes to read • Edit Online

Les *types numériques à virgule flottante* représentent des nombres réels. Tous les types numériques à virgule flottante sont des types valeur. Il s'agit également de types simples qui peuvent être initialisés avec des littéraux. Tous les types numériques à virgule flottante prennent en charge les opérateurs arithmétiques, de comparaisonet d' égalité .

Caractéristiques des types à virgule flottante

C# prend en charge les types à virgule flottante prédéfinis suivants :

C# TYPE/MOT CLÉ	PLAGE APPROXIMATIVE	PRECISION	TAILLE	TYPE .NET
`float`	$\pm 1{,}5 \times 10^{-45}$ à $\pm 3{,}4 \times 10^{38}$	~6-9 chiffres	4 octets	System.Single
`double`	De $\pm 5{,}0 \times 10^{-324}$ à $\pm 1{,}7 \times 10^{308}$	~15-17 chiffres	8 octets	System.Double
`decimal`	$\pm 1{,}0 \times 10^{-28}$ to $\pm 7{,}9228 \times 10^{28}$	28 à 29 chiffres	16 octets	System.Decimal

Dans le tableau précédent, chaque mot clé de type C# de la colonne la plus à gauche est un alias pour le type .NET correspondant. Ils sont interchangeables. Par exemple, les déclarations suivantes déclarent des variables du même type :

```
double a = 12.3;
System.Double b = 12.3;
```

La valeur par défaut pour tous les types virgule flottante est zéro, `0`. Chacun des types à virgule flottante a les constantes `MinValue` et `MaxValue` qui fournissent la valeur finie minimale et maximale de ce type. Les types `float` et `double` fournissent également des constantes qui représentent des valeurs NaN (Not-a-Number) et d'infini. Par exemple, le type `double` fournit les constantes suivantes : Double.NaN, Double.NegativeInfinity et Double.PositiveInfinity.

Le type `decimal` fournit une plus grande précision et une plage de valeurs plus petite que `float` et `double`, ce qui le rend particulièrement approprié aux calculs financiers et monétaires.

Vous pouvez mélanger les types intégraux et les `float` `double` types et dans une expression. Dans ce cas, les types intégraux sont convertis implicitement en un des types à virgule flottante et, si nécessaire, le `float` type est implicitement converti en `double` . L'expression est évaluée de la manière suivante :

- S'il existe `double` un type dans l'expression, l'expression prend la valeur `double` , ou `bool` dans les comparaisons relationnelles et d'égalité.
- S'il n'y a aucun `double` type dans l'expression, l'expression prend la valeur `float` , ou `bool` dans les comparaisons relationnelles et d'égalité.

Vous pouvez également mélanger des types intégraux et le `decimal` type dans une expression. Dans ce cas, les types intégraux sont convertis implicitement vers le `decimal` type et l'expression prend la valeur `decimal` , ou `bool` dans les comparaisons relationnelles et d'égalité.

Vous ne pouvez pas mélanger le `decimal` type `float` aux `double` types et dans une expression. Dans ce cas, si vous souhaitez effectuer des opérations arithmétiques, de comparaison ou d'égalité, vous devez convertir explicitement les opérandes à partir de ou vers le `decimal` type, comme le montre l'exemple suivant :

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

Vous pouvez utiliser des chaînes au format numérique standard ou des chaînes au format numérique personnalisées pour mettre en forme une valeur à virgule flottante.

Littéraux réels

Le type d'un littéral réel est déterminé par son suffixe comme suit :

- Le littéral sans suffixe ou avec `d` le `D` suffixe ou est de type `double`
- Le littéral avec le `f` `F` suffixe ou est de type `float`
- Le littéral avec le `m` `M` suffixe ou est de type `decimal`

Le code suivant illustre un exemple de chaque :

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

L'exemple précédent illustre également l'utilisation de `_` comme *séparateur de chiffres*, qui est pris en charge à partir de C# 7,0. Vous pouvez utiliser le séparateur de chiffres avec tous les types de littéraux numériques.

Vous pouvez également utiliser la notation scientifique, c'est-à-dire spécifier une partie exposant d'un littéral réel, comme le montre l'exemple suivant :

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

Conversions

Il n'existe qu'une seule conversion implicite entre les types numériques à virgule flottante : de `float` à

`double` . Toutefois, vous pouvez convertir n'importe quel type à virgule flottante en tout autre type à virgule flottante avec le cast explicite. Pour plus d'informations, consultez conversions numériques intégrées.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types virgule flottante
- Type décimal
- Littéraux réels

Voir aussi

- Informations de référence sur C#
- Types valeur
- Types intégraux
- Chaînes de format numériques standard
- Valeurs numériques dans .NET
- System.Numerics.Complex

Conversions numériques intégrées (référence C#)

02/11/2020 • 9 minutes to read • Edit Online

C# fournit un ensemble de types numériques intégraux et à virgule flottante . Il existe une conversion entre deux types numériques, implicite ou explicite. Vous devez utiliser une expression de cast pour effectuer une conversion explicite.

Conversions numériques implicites

Le tableau suivant montre les conversions implicites prédéfinies entre les types numériques intégrés :

À PARTIR	À
sbyte	`short` , `int` , `long` , `float` , `double` ou `decimal`
byte	`short` , `ushort` , `int` , `uint` , `long` , `ulong` , `float` , `double` ou `decimal`
short	`int` , `long` , `float` , `double` ou `decimal`
ushort	`int` , `uint` , `long` , `ulong` , `float` , `double` ou `decimal`
int	`long` , `float` , `double` ou `decimal`
uint	`long` , `ulong` , `float` , `double` ou `decimal`
long	`float` , `double` ou `decimal`
ulong	`float` , `double` ou `decimal`
float	`double`

NOTE

Les conversions implicites de `int` ,, `uint` `long` ou `ulong` vers `float` et à partir de `long` ou `ulong` en `double` peuvent entraîner une perte de précision, mais jamais une perte d'un ordre de grandeur. Les autres conversions numériques implicites ne perdent jamais d'informations.

Notez également que

- Tout type numérique intégral est implicitement convertible en type numérique à virgule flottante.
- Il n'existe aucune conversion implicite des `byte` types et `sbyte` . Il n'existe aucune conversion implicite à partir des types `double` et `decimal` .
- Il n'existe aucune conversion implicite entre le type `decimal` et le type `float` ou `double` .
- La valeur d'une expression constante de type `int` (par exemple, une valeur représentée par un littéral entier) peut être implicitement convertie en `sbyte` , `byte` , `short` , `ushort` , `uint` ou `ulong` , si elle se

trouve dans la plage du type de destination :

```
byte a = 13;
byte b = 300;  // CS0031: Constant value '300' cannot be converted to a 'byte'
```

Comme le montre l'exemple précédent, si la valeur de constante n'est pas comprise dans la plage du type de destination, une erreur de compilateur CS0031 se produit.

Conversions numériques explicites

Le tableau suivant montre les conversions explicites prédéfinies entre les types numériques intégrés pour lesquels il n'existe pas de conversion implicite:

À PARTIR	À
sbyte	`byte`, `ushort`, `uint` ou `ulong`
byte	`sbyte`
short	`sbyte`, `byte`, `ushort`, `uint` ou `ulong`
ushort	`sbyte`, `byte` ou `short`
int	`sbyte`, `byte`, `short`, `ushort`, `uint` ou `ulong`
uint	`sbyte`, `byte`, `short`, `ushort` ou `int`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint` ou `ulong`
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint` ou `long`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` ou `decimal`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` ou `decimal`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` ou `double`

NOTE

Une conversion numérique explicite peut entraîner une perte de données ou lever une exception, en général un OverflowException .

Notez également que

- Quand vous convertissez une valeur d'un type intégral en un autre type intégral, le résultat varie selon le contexte de vérification du dépassement. Dans un contexte vérifié, la conversion réussit si la valeur source se situe dans la plage du type de destination. Sinon, une exception OverflowException est levée. Dans un contexte non vérifié, la conversion réussit toujours et procède comme suit :
  - Si le type source est plus grand que le type de destination, la valeur source est tronquée via

l'abandon de ses bits « supplémentaires » les plus significatifs. Le résultat est ensuite traité en tant que valeur du type de destination.

  - Si le type de source est plus petit que le type de destination, la valeur source est un signe étendu ou étendu à zéro, afin qu'elle soit de la même taille que le type de destination. L'extension de signe est utilisée si le type source est signé. L'extension de zéro est utilisée si le type source est non signé. Le résultat est ensuite traité en tant que valeur du type de destination.

  - Si le type source a la même taille que le type de destination, la valeur source est traitée comme une valeur du type de destination.

- Quand vous convertissez une valeur de type `decimal` en type intégral, la valeur source est arrondie vers zéro à la valeur intégrale la plus proche. Si la valeur intégrale obtenue se trouve en dehors de la plage du type de destination, une exception OverflowException est levée.

- Quand vous convertissez une valeur `double` ou `float` en type intégral, cette valeur est arrondie vers zéro, à la valeur intégrale la plus proche. Si la valeur intégrale résultante se situe en dehors de la plage du type de destination, le résultat varie selon le contexte de vérification du dépassement. Dans un contexte vérifié, une exception OverflowException est levée, alors que dans un contexte non vérifié, le résultat est une valeur non spécifiée du type de destination.

- Quand vous convertissez `double` en `float`, la valeur `double` est arrondie à la valeur `float` la plus proche. Si la `double` valeur est trop petite ou trop grande pour être contenue dans le `float` type, le résultat est zéro ou infini.

- Dans le cas d'une conversion de `float` ou `double` en `decimal`, la valeur source est convertie en une représentation `decimal` et arrondie au nombre le plus proche après la 28ème décimale, si nécessaire. Selon la valeur de la valeur source, la conversion peut aboutir à l'un des résultats suivants :

  - Si la valeur source est trop petite pour être représentée en tant que `decimal`, le résultat est zéro.

  - Si la valeur source est une valeur non numérique (NaN), un nombre infini ou une valeur trop grande pour être représentée au format `decimal`, une exception OverflowException est levée.

- Lorsque vous convertissez `decimal` en `float` ou `double` , la valeur source est arrondie à la `float` valeur ou la plus proche `double` , respectivement.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Conversions numériques implicites
- Conversions numériques explicites

Voir aussi

- Informations de référence sur C#
- Cast et conversions de types

bool (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le `bool` mot clé type est un alias pour le System.Boolean type de structure .net qui représente une valeur booléenne, qui peut être `true` ou `false` .

Pour effectuer des opérations logiques avec des valeurs de `bool` type, utilisez des opérateurs logiques booléens . Le `bool` type est le type de résultat des opérateurs de comparaison et d' égalité . Une `bool` expression peut être une expression conditionnelle de contrôle dans les instructions If, do, whileet for et dans l' opérateur `?:` conditionnel .

La valeur par défaut du `bool` type est `false` .

Littéraux

Vous pouvez utiliser les `true` `false` littéraux et pour initialiser une `bool` variable ou pour passer une `bool` valeur :

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Logique booléenne à trois valeurs

Utilisez le `bool?` type Nullable, si vous avez besoin de prendre en charge la logique à trois valeurs, par exemple, lorsque vous utilisez des bases de données qui prennent en charge un type booléen à trois valeurs. Dans le cas des opérandes `bool?`, les opérateurs prédéfinis `&` et `|` prennent en charge la logique à trois valeurs. Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Pour plus d'informations sur les types valeur Nullable, consultez types valeur Nullable.

Conversions

C# fournit uniquement deux conversions qui impliquent le `bool` type. Il s'agit d'une conversion implicite vers le type Nullable correspondant `bool?` et d'une conversion explicite du `bool?` type. Toutefois, .NET fournit des méthodes supplémentaires que vous pouvez utiliser pour convertir vers ou à partir du `bool` type. Pour plus d'informations, consultez la section conversion de valeurs booléennes en et à partir de la System.Boolean page de référence des API.

spécification du langage C#

Pour plus d'informations, consultez la section type bool de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- opérateurs true et false

Char (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le `char` mot clé type est un alias pour le System.Char type de structure .net qui représente un caractère Unicode UTF-16.

TYPE	PLAGE	TAILLE	TYPE .NET
`char`	U+0000 à U+FFFF	16 bits	System.Char

La valeur par défaut du `char` type est `\0` , c'est-à-dire U + 0000.

Le `char` type prend en charge les opérateurs de comparaison, d' égalité, d' incrémentationet de décrémentation . En outre, pour les `char` opérandes, les opérateurs logiques arithmétiques et au niveau du bit effectuent une opération sur les codes de caractères correspondants et produisent le résultat du `int` type.

Le type String représente du texte sous la forme d'une séquence de `char` valeurs.

Littéraux

Vous pouvez spécifier une `char` valeur avec :

- littéral de caractère.
- séquence d'échappement Unicode, suivie de `\u` la représentation hexadécimale à quatre symboles d'un code de caractère.
- séquence d'échappement hexadécimale, `\x` suivie de la représentation hexadécimale d'un code de caractère.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

Comme le montre l'exemple précédent, vous pouvez également effectuer un cast de la valeur d'un code de caractère dans la `char` valeur correspondante.

> **NOTE**
>
> Dans le cas d'une séquence d'échappement Unicode, vous devez spécifier les quatre chiffres hexadécimaux. Autrement dit, `\u006A` est une séquence d'échappement valide, tandis que `\u06A` et `\u6A` ne sont pas valides.
>
> Dans le cas d'une séquence d'échappement hexadécimale, vous pouvez omettre les zéros non significatifs. Autrement dit, les `\x006A` `\x06A` `\x6A` séquences d'échappement, et sont valides et correspondent au même caractère.

Conversions

Le `char` type est implicitement convertible en types intégraux suivants : `ushort` , `int` , `uint` , `long` et

`ulong` . Il est également implicitement convertible en types numériques à virgule flottante intégrés : `float` , `double` et `decimal` . Il est explicitement convertible en `sbyte` `byte` types, et en `short` types intégraux.

Il n'existe pas de conversion implicite d'autres types vers le `char` type. Toutefois, tous les types numériques intégraux ou à virgule flottante sont explicitement convertibles en `char` .

spécification du langage C#

Pour plus d'informations, consultez la section types intégraux de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Chaînes
- System.Text.Rune
- Encodage de caractères dans .NET

Types énumération (référence C#)

16/12/2020 • 6 minutes to read • Edit Online

Un type d' *énumération* (ou un *type enum*) est un type valeur défini par un ensemble de constantes nommées du type numérique intégral sous-jacent. Pour définir un type d'énumération, utilisez le `enum` mot clé et spécifiez les noms des *membres enum*:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

Par défaut, les valeurs constantes associées des membres enum sont de type `int` ; elles commencent par zéro et augmentent d'une unité après l'ordre du texte de définition. Vous pouvez spécifier explicitement n'importe quel autre type numérique intégral en tant que type sous-jacent d'un type énumération. Vous pouvez également spécifier explicitement les valeurs de constante associées, comme le montre l'exemple suivant :

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Vous ne pouvez pas définir une méthode à l'intérieur de la définition d'un type d'énumération. Pour ajouter des fonctionnalités à un type énumération, créez une méthode d'extension.

La valeur par défaut d'un type énumération `E` est la valeur produite par expression `(E)0`, même si zéro n'a pas le membre enum correspondant.

Vous utilisez un type d'énumération pour représenter un choix à partir d'un ensemble de valeurs s'excluant mutuellement ou d'une combinaison de choix. Pour représenter une combinaison de choix, définissez un type d'énumération en tant qu'indicateurs binaires.

Types énumération comme indicateurs binaires

Si vous souhaitez qu'un type énumération représente une combinaison de choix, définissez des membres enum pour ces choix de manière à ce qu'un choix individuel soit un champ de bits. Autrement dit, les valeurs associées à ces membres enum doivent être les puissances de deux. Ensuite, vous pouvez utiliser les opérateurs logiques au niveau du bit `|` `&` ou combiner des choix ou croiser des combinaisons de choix, respectivement. Pour indiquer qu'un type énumération déclare des champs de bits, appliquez l'attribut Flags à celui-ci. Comme le montre l'exemple suivant, vous pouvez également inclure des combinaisons typiques dans la définition d'un type énumération.

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

Pour obtenir plus d'informations et des exemples, consultez la System.FlagsAttribute page de référence de l'API et la section membres non exclusifs et l'attribut flags de la System.Enum page de référence des API.

Le type System. Enum et la contrainte enum

Le System.Enum type est la classe de base abstraite de tous les types énumération. Il fournit un certain nombre de méthodes pour obtenir des informations sur un type d'énumération et ses valeurs. Pour plus d'informations et d'exemples, consultez la System.Enum page de référence des API.

À compter de C# 7,3, vous pouvez utiliser `System.Enum` dans une contrainte de classe de base (appelée contrainte enum) pour spécifier qu'un paramètre de type est un type énumération. Tout type énumération satisfait également `struct` à la contrainte, qui est utilisée pour spécifier qu'un paramètre de type est un type valeur qui n'autorise pas les valeurs NULL.

Conversions

Pour tout type énumération, il existe des conversions explicites entre le type énumération et son type intégral sous-jacent. Si vous effectuez un cast d'une valeur enum en son type sous-jacent, le résultat est la valeur intégrale associée d'un membre Enum.

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output: Integral value of
Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Utilisez la Enum.IsDefined méthode pour déterminer si un type énumération contient un membre enum avec la certaine valeur associée.

Pour tout type énumération, il existe des conversions boxing et unboxing vers et à partir du System.Enum type, respectivement.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Énumérations
- Valeurs et opérations des énumérations
- Opérateurs logiques d'énumération
- Opérateurs de comparaison d'énumération
- Conversions d'énumération explicites
- Conversions d'énumération implicites

Voir aussi

- Informations de référence sur C#
- Chaînes de format d'énumération
- Directives de conception-conception enum
- Directives de conception-enum Naming conventions
- Instruction switch

Types de structures (référence C#)

16/12/2020 • 14 minutes to read • Edit Online

Un type de *structure* (ou *type struct*) est un type valeur qui peut encapsuler des données et des fonctionnalités associées. Le `struct` mot clé vous permet de définir un type de structure :

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Les types de structure ont une *sémantique de valeur*. Autrement dit, une variable d'un type structure contient une instance du type. Par défaut, les valeurs des variables sont copiées lors de l'assignation, en passant un argument à une méthode et en retournant un résultat de méthode. Dans le cas d'une variable de type structure, une instance du type est copiée. Pour plus d'informations, consultez types valeur.

En général, vous utilisez des types de structure pour concevoir de petits types centrés sur les données qui fournissent peu ou pas de comportement. Par exemple, .NET utilise des types structure pour représenter un nombre (entier et réel), une valeur booléenne, un caractère Unicode, une instance d'heure. Si vous avez le focus sur le comportement d'un type, vous pouvez définir une classe. Les types de classe ont une *sémantique de référence*. Autrement dit, une variable d'un type de classe contient une référence à une instance du type, et non à l'instance elle-même.

Étant donné que les types de structure ont une sémantique de valeur, nous vous recommandons de définir des types de structures *immuables* .

`readonly` modélis

À compter de C# 7,2, vous utilisez le `readonly` modificateur pour déclarer qu'un type structure est immuable :

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

Tous les membres de données d'un `readonly` struct doivent être en lecture seule, comme suit :

- Toute déclaration de champ doit avoir le `readonly` modificateur
- Toute propriété, y compris celles implémentées automatiquement, doit être en lecture seule. Dans C# 9,0 et versions ultérieures, une propriété peut avoir un `init` accesseur.

Cela garantit qu'aucun membre d'un `readonly` struct ne modifie l'état de la structure. En C# 8,0 et versions ultérieures, cela signifie que d'autres membres d'instance, à l'exception des constructeurs, sont implicitement `readonly` .

> **NOTE**
> Dans un `readonly` struct, un membre de données d'un type référence mutable peut toujours muter son propre État. Par exemple, vous ne pouvez pas remplacer une List<T> instance, mais vous pouvez lui ajouter de nouveaux éléments.

`readonly` membres d'instance

À compter de C# 8,0, vous pouvez également utiliser le `readonly` modificateur pour déclarer qu'un membre d'instance ne modifie pas l'état d'un struct. Si vous ne pouvez pas déclarer le type de structure entier comme `readonly` , utilisez le `readonly` modificateur pour marquer les membres d'instance qui ne modifient pas l'état de la structure.

Dans un `readonly` membre d'instance, vous ne pouvez pas assigner aux champs d'instance de la structure. Toutefois, un `readonly` membre peut appeler un non- `readonly` membre. Dans ce cas, le compilateur crée une copie de l'instance de structure et appelle le non `readonly` membre sur cette copie. Par conséquent, l'instance de la structure d'origine n'est pas modifiée.

En règle générale, vous appliquez le `readonly` modificateur aux genres suivants de membres d'instance :

- leurs

  ```
  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  Vous pouvez également appliquer le `readonly` modificateur aux méthodes qui

remplacent les méthodes déclarées dans System.Object :

```
public readonly override string ToString() => $"({X}, {Y})";
```

- Propriétés et indexeurs :

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

Si vous devez appliquer le `readonly` modificateur aux deux accesseurs d'une propriété ou d'un indexeur, appliquez-le dans la déclaration de la propriété ou de l'indexeur.

> **NOTE**
> Le compilateur déclare un `get` accesseur d'une propriété implémentée automatiquement en tant que `readonly` , quelle que soit la présence du `readonly` modificateur dans une déclaration de propriété.

Dans C# 9,0 et versions ultérieures, vous pouvez appliquer le `readonly` modificateur à une propriété ou à un indexeur avec un `init` accesseur :

```
public readonly double X { get; init; }
```

Vous ne pouvez pas appliquer le `readonly` modificateur aux membres statiques d'un type structure.

Le compilateur peut utiliser le `readonly` modificateur pour les optimisations de performances. Pour plus d'informations, consultez écrire du code C# sécurisé et efficace.

Limitations liées à la conception d'un type structure

Lorsque vous concevez un type structure, vous avez les mêmes fonctionnalités qu'avec un type de classe , avec les exceptions suivantes :

- Vous ne pouvez pas déclarer un constructeur sans paramètre. Chaque type de structure fournit déjà un constructeur sans paramètre implicite qui produit la valeur par défaut du type.
- Vous ne pouvez pas initialiser un champ ou une propriété d'instance dans sa déclaration. Toutefois, vous pouvez initialiser un champ statique ou const ou une propriété statique dans sa déclaration.
- Un constructeur d'un type structure doit initialiser tous les champs d'instance du type.
- Un type structure ne peut pas hériter d'un autre type de classe ou de structure et il ne peut pas être la base d'une classe. Toutefois, un type structure peut implémenter des interfaces.
- Vous ne pouvez pas déclarer un finaliseur dans un type structure.

Instanciation d'un type structure

En C#, vous devez initialiser une variable déclarée avant de pouvoir l'utiliser. Comme une variable de type structure ne peut pas être `null` (à moins qu'il ne s'agisse d'une variable d'un type valeur Nullable), vous devez instancier une instance du type correspondant. Il existe plusieurs façons de procéder.

En général, vous instanciez un type structure en appelant un constructeur approprié avec l' `new` opérateur. Chaque type de structure a au moins un constructeur. Il s'agit d'un constructeur sans paramètre implicite, qui produit la valeur par défaut du type. Vous pouvez également utiliser une expression de valeur par défaut pour produire la valeur par défaut d'un type.

Si tous les champs d'instance d'un type structure sont accessibles, vous pouvez également l'instancier sans l' `new` opérateur. Dans ce cas, vous devez initialiser tous les champs d'instance avant la première utilisation de l'instance. L'exemple suivant montre comment effectuer cette opération :

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

Dans le cas des types valeur intégrés, utilisez les littéraux correspondants pour spécifier une valeur du type.

Passage de variables de type structure par référence

Lorsque vous transmettez une variable de type structure à une méthode en tant qu'argument ou que vous retournez une valeur structure-type à partir d'une méthode, l'ensemble de l'instance d'un type structure est copié. Cela peut affecter les performances de votre code dans les scénarios de haute performance qui impliquent des types de structure volumineux. Vous pouvez éviter la copie de valeurs en passant une variable de type structure par référence. Utilisez les `ref` `out` `in` modificateurs de paramètre de méthode, ou pour indiquer qu'un argument doit être passé par référence. Utilisez les retours Ref pour retourner un résultat de méthode par référence. Pour plus d'informations, consultez écrire du code C# sécurisé et efficace.

`ref` modélis

À compter de C# 7,2, vous pouvez utiliser le `ref` modificateur dans la déclaration d'un type structure. Les instances d'un `ref` type struct sont allouées sur la pile et ne peuvent pas échapper au tas managé. Pour garantir que, le compilateur limite l'utilisation des `ref` types struct comme suit :

- Un `ref` struct ne peut pas être le type d'élément d'un tableau.
- Un `ref` struct ne peut pas être un type déclaré d'un champ d'une classe ou d'un non-`ref` struct.
- Un `ref` struct ne peut pas implémenter des interfaces.
- Un `ref` struct ne peut pas être converti en System.ValueType ou System.Object .
- Un `ref` struct ne peut pas être un argument de type.
- Une `ref` variable de struct ne peut pas être capturée par une expression lambda ou une fonction locale.
- Une `ref` variable de struct ne peut pas être utilisée dans une `async` méthode. Toutefois, vous pouvez utiliser des `ref` variables de struct dans des méthodes synchrones, par exemple, dans celles qui retournent Task ou Task<TResult> .
- Une `ref` variable de struct ne peut pas être utilisée dans les itérateurs.

En général, vous définissez un `ref` type struct lorsque vous avez besoin d'un type qui comprend également des membres de données de `ref` types struct :

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

Pour déclarer un `ref` struct en tant que `readonly` , combinez les `readonly` `ref` modificateurs et dans la déclaration de type (le `readonly` modificateur doit précéder le `ref` modificateur) :

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

Dans .NET, les exemples d'un `ref` struct sont System.Span<T> et System.ReadOnlySpan<T> .

contrainte de struct

Vous utilisez également le `struct` mot clé dans la `struct` contrainte pour spécifier qu'un paramètre de type est un type valeur qui n'autorise pas les valeurs NULL. Les types de structure et d' énumération satisfont à la `struct` contrainte.

Conversions

Pour tout type de structure (à l'exception des types `ref` struct), il existe des conversions boxing et unboxing vers et à partir des System.ValueType System.Object types et. Il existe également des conversions boxing et unboxing entre un type structure et toute interface qu'il implémente.

spécification du langage C#

Pour plus d’informations, consultez la section structs de la spécification du langage C#.

Pour plus d’informations sur les fonctionnalités introduites dans C# 7,2 et versions ultérieures, consultez les notes de proposition de fonctionnalités suivantes :

- Structs en lecture seule
- Membres d'instance en lecture seule
- Sécurité au moment de la compilation pour les types référence

Voir aussi

- Informations de référence sur C#
- Instructions de conception-choix entre une classe et une structure
- Directives de conception-conception de struct
- Classes et structs

Types de tuples (référence C#)

02/11/2020 • 13 minutes to read • Edit Online

Disponible en C# 7,0 et versions ultérieures, la fonctionnalité de *tuples* fournit une syntaxe concise pour regrouper plusieurs éléments de données dans une structure de données légère. L'exemple suivant montre comment vous pouvez déclarer une variable de tuple, l'initialiser et accéder à ses membres de données :

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Comme le montre l'exemple précédent, pour définir un type de tuple, vous spécifiez les types de tous ses membres de données et, éventuellement, les noms de champs. Vous ne pouvez pas définir de méthodes dans un type de tuple, mais vous pouvez utiliser les méthodes fournies par .NET, comme le montre l'exemple suivant :

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

À compter de C# 7,3, les types de tuple prennent en charge les opérateurs d'égalité `==` et `!=` . Pour plus d'informations, consultez la section égalité des tuples .

Les types de tuples sont des types valeur; les éléments de tuple sont des champs publics. Cela rend les types valeur *mutable* de tuples.

> **NOTE**
>
> La fonctionnalité des tuples requiert le System.ValueTuple type et les types génériques associés (par exemple, System.ValueTuple<T1,T2>), qui sont disponibles dans .net Core et .NET Framework 4,7 et versions ultérieures. Pour utiliser des tuples dans un projet qui cible .NET Framework 4.6.2 ou version antérieure, ajoutez le package NuGet `System.ValueTuple` au projet.

Vous pouvez définir des tuples avec un grand nombre d'éléments arbitraires :

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26);  // output: 26
```

Cas d'usage de tuples

L'un des cas d'utilisation les plus courants de tuples est le type de retour de la méthode. Autrement dit, au lieu de définir des `out` paramètres de méthode, vous pouvez regrouper les résultats de méthode dans un type de retour de tuple, comme le montre l'exemple suivant :

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

Comme le montre l'exemple précédent, vous pouvez utiliser l'instance de tuple retournée directement ou la décomposer dans des variables distinctes.

Vous pouvez également utiliser des types de tuples au lieu de types anonymes. par exemple, dans les requêtes LINQ. Pour plus d'informations, consultez choix entre les types de tuples et anonymes.

En général, vous utilisez des tuples pour regrouper des éléments de données faiblement liés. Cela est généralement utile dans les méthodes utilitaires privées et internes. Dans le cas d'une API publique, envisagez de définir une classe ou un type de structure .

Noms de champs de Tuple

Vous pouvez spécifier explicitement les noms des champs de tuple dans une expression d'initialisation de tuple ou dans la définition d'un type de tuple, comme le montre l'exemple suivant :

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

À compter de C# 7,1, si vous ne spécifiez pas de nom de champ, il peut être déduit à partir du nom de la

variable correspondante dans une expression d'initialisation de tuple, comme le montre l'exemple suivant :

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

C'est ce que l'on appelle des initialiseurs de projection de Tuple. Le nom d'une variable n'est pas projeté sur un nom de champ de tuple dans les cas suivants :

- Le nom du candidat est un nom de membre d'un type de tuple, par exemple,, `Item3` `ToString` ou `Rest` .
- Le nom du candidat est un doublon d'un autre nom de champ de tuple, explicite ou implicite.

Dans ce cas, vous spécifiez explicitement le nom d'un champ ou vous accédez à un champ par son nom par défaut.

Les noms par défaut des champs de tuple sont `Item1` , `Item2` , `Item3` et ainsi de suite. Vous pouvez toujours utiliser le nom par défaut d'un champ, même lorsqu'un nom de champ est spécifié de manière explicite ou déduite, comme le montre l'exemple suivant :

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

Les comparaisons d'égalité des tuples et des tuples ne prennent pas en compte les noms de champs.

Au moment de la compilation, le compilateur remplace les noms de champs non par défaut par les noms par défaut correspondants. Par conséquent, les noms de champs explicitement spécifiés ou inférés ne sont pas disponibles au moment de l'exécution.

Assignation et déconstruction de Tuple

C# prend en charge l'assignation entre les types tuple qui satisfont les deux conditions suivantes :

- les deux types de tuples ont le même nombre d'éléments
- pour chaque position de tuple, le type de l'élément de tuple de droite est le même que ou implicitement convertible en type de l'élément de tuple de gauche correspondant

Les valeurs d'éléments tuples sont assignées à la suite de l'ordre des éléments de Tuple. Les noms des champs de tuple sont ignorés et non assignés, comme le montre l'exemple suivant :

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

Vous pouvez également utiliser l'opérateur `=` d'assignation pour *déconstruire* une instance de tuple dans des variables distinctes. Pour ce faire, vous pouvez procéder de l'une des façons suivantes :

- Déclarez explicitement le type de chaque variable à l'intérieur des parenthèses :

  ```
  var t = ("post office", 3.6);
  (string destination, double distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

- Utilisez le `var` mot clé en dehors des parenthèses pour déclarer les variables implicitement typées et laisser le compilateur déduire leurs types :

  ```
  var t = ("post office", 3.6);
  var (destination, distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

- Utiliser des variables existantes :

  ```
  var destination = string.Empty;
  var distance = 0.0;

  var t = ("post office", 3.6);
  (destination, distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

Pour plus d'informations sur la déconstruction de tuples et d'autres types, consultez déconstruction de tuples et d'autres types.

Égalité des tuples

À compter de C# 7.3, les types tuple prennent en charge les opérateurs `==` et `!=`. Ces opérateurs comparent les membres de l'opérande de gauche avec les membres correspondants de l'opérande de droite suivant l'ordre des éléments de Tuple.

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

Comme le montre l'exemple précédent, `==` les `!=` opérations et ne prennent pas en compte les noms de champs de Tuple.

Deux tuples sont comparables lorsque les deux conditions suivantes sont satisfaites :

- Les deux tuples ont le même nombre d'éléments. Par exemple, `t1 != t2` ne compile pas si `t1` et `t2` ont un nombre d'éléments différent.
- Pour chaque position de tuple, les éléments correspondants des opérandes de tuple de gauche et de droite sont comparables aux `==` opérateurs et `!=` . Par exemple, `(1, (2, 3)) == ((1, 2), 3)` ne se compile `1` pas, car n'est pas comparable à `(1, 2)` .

Les `==` `!=` opérateurs et comparent les tuples en mode court-circuit. Autrement dit, une opération s'arrête dès qu'elle rencontre une paire d'éléments non égaux ou atteint les terminaisons des tuples. Toutefois, avant toute comparaison, *tous les* éléments tuples sont évalués, comme le montre l'exemple suivant :

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False
```

Tuples en tant que paramètres de sortie

En règle générale, vous refactorisez une méthode qui a des `out` paramètres dans une méthode qui retourne un tuple. Toutefois, il existe des cas dans lesquels un `out` paramètre peut être de type Tuple. L'exemple suivant montre comment utiliser des tuples en tant que `out` Paramètres :

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is {limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

Tuples et `System.Tuple`

Les tuples C#, qui sont sauvegardés par les System.ValueTuple types, sont différents des tuples représentés par les System.Tuple types. Les principales différences sont les suivantes :

- `ValueTuple` les types sont des types valeur. `Tuple` les types sont des types référence.
- `ValueTuple` les types sont mutables. `Tuple` les types sont immuables.
- Les membres de données de `ValueTuple` types sont des champs. Les membres de données de `Tuple` types sont des propriétés.

spécification du langage C#

Pour plus d'informations, consultez les notes de la proposition de fonctionnalités suivantes :

- Déduire les noms de tuples (également appelés initialiseurs de projection de Tuple)
- Prise en charge pour `==` et `!=` sur les types de Tuple

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Choix entre les types de tuples et anonymes
- System.ValueTuple

Types valeur Nullable (référence C#)

02/11/2020 • 13 minutes to read • Edit Online

Un *type valeur Nullable* `T?` représente toutes les valeurs de son type valeur sous-jacent `T` et une valeur null supplémentaire. Par exemple, vous pouvez assigner l'une des trois valeurs suivantes à une `bool?` variable : `true` , `false` ou `null` . Un type valeur sous-jacent `T` ne peut pas être un type valeur Nullable lui-même.

> **NOTE**
> C# 8,0 introduit la fonctionnalité de types référence Nullable. Pour plus d'informations, consultez types de référence Nullable. Les types valeur Nullable sont disponibles à partir de C# 2.

Tout type valeur Nullable est une instance de la System.Nullable<T> structure générique. Vous pouvez faire référence à un type valeur Nullable avec un type sous-jacent `T` dans l'un des formulaires interchangeables suivants : `Nullable<T>` ou `T?` .

En général, vous utilisez un type valeur Nullable lorsque vous devez représenter la valeur indéfinie d'un type valeur sous-jacent. Par exemple, une variable booléenne, ou `bool` , ne peut être que `true` ou `false` . Toutefois, dans certaines applications, une valeur de variable peut être indéfinie ou manquante. Par exemple, un champ de base de données peut contenir `true` ou `false` , ou il ne peut contenir aucune valeur, autrement dit, `NULL` . Vous pouvez utiliser le `bool?` type dans ce scénario.

Déclaration et affectation

Comme un type valeur est implicitement convertible en type valeur Nullable correspondant, vous pouvez assigner une valeur à une variable d'un type valeur Nullable comme vous le feriez pour son type valeur sous-jacent. Vous pouvez également affecter la `null` valeur. Par exemple :

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

La valeur par défaut d'un type valeur Nullable représente `null` , autrement dit, une instance dont la Nullable<T>.HasValue propriété retourne `false` .

Examen d'une instance d'un type valeur Nullable

À compter de C# 7,0, vous pouvez utiliser l' `is` opérateur avec un modèle de type pour examiner à la fois une instance d'un type de valeur Nullable pour `null` et récupérer une valeur d'un type sous-jacent :

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Vous pouvez toujours utiliser les propriétés en lecture seule suivantes pour examiner et obtenir la valeur d'une variable de type valeur Nullable :

- Nullable<T>.HasValue indique si une instance d'un type valeur Nullable a une valeur de son type sous-jacent.
- Nullable<T>.Value obtient la valeur d'un type sous-jacent si HasValue est `true`. Si HasValue est `false`, la propriété Value lève une InvalidOperationException.

L'exemple suivant utilise la `HasValue` propriété pour déterminer si la variable contient une valeur avant de l'afficher :

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

Vous pouvez également comparer une variable d'un type valeur Nullable avec `null` au lieu d'utiliser la `HasValue` propriété, comme le montre l'exemple suivant :

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversion d'un type valeur Nullable en un type sous-jacent

Si vous souhaitez assigner une valeur d'un type valeur Nullable à une variable de type valeur n'acceptant pas les valeurs NULL, vous devrez peut-être spécifier la valeur à assigner à la place de `null`. Utilisez l' opérateur `??` de fusion Null pour effectuer cette opération (vous pouvez également utiliser la Nullable<T>.GetValueOrDefault(T) méthode dans le même but) :

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

Si vous souhaitez utiliser la valeur par défaut du type valeur sous-jacent à la place de `null`, utilisez la Nullable<T>.GetValueOrDefault() méthode.

Vous pouvez également effectuer un cast explicite d'un type valeur Nullable vers un type non Nullable, comme le montre l'exemple suivant :

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

Au moment de l'exécution, si la valeur d'un type valeur Nullable est `null`, le cast explicite lève une InvalidOperationException .

Un type valeur n'acceptant pas les valeurs null `T` est implicitement convertible en type valeur Nullable correspondant `T?` .

Opérateurs levés

Les opérateurs unaires et binaires prédéfinis ou tous les opérateurs surchargés pris en charge par un type valeur `T` sont également pris en charge par le type valeur Nullable correspondant `T?` . Ces opérateurs, également appelés *opérateurs levés*, produisent `null` si l'un des opérandes ou les deux sont `null` ; sinon, l'opérateur utilise les valeurs contenues de ses opérandes pour calculer le résultat. Par exemple :

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

> **NOTE**
> Pour le `bool?` type, les opérateurs prédéfinis `&` et `|` ne suivent pas les règles décrites dans cette section : le résultat d'une évaluation d'opérateur peut être non null, même si l'un des opérandes est `null` . Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Pour les opérateurs de comparaison `<` , `>` , `<=` et `>=` , si l'un des opérandes ou les deux sont `null` , le résultat est `false` ; sinon, les valeurs contenues des opérandes sont comparées. Ne partez pas du principe que parce qu'une comparaison spécifique (par exemple `<=`) retourne `false` , la comparaison opposée (`>`) retourne `true` . L'exemple suivant montre que 10 n'est

- ni supérieur ou égal à `null`
- ni inférieur à `null`

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

Pour l' opérateur d'égalité `==` , si les deux opérandes sont `null` , le résultat est `true` , si un seul des opérandes est `null` , le résultat est `false` ; sinon, les valeurs contenues des opérandes sont comparées.

Pour l' opérateur d'inégalité `!=` , si les deux opérandes sont `null` , le résultat est `false` , si un seul des opérandes est `null` , le résultat est `true` ; sinon, les valeurs contenues des opérandes sont comparées.

S'il existe une conversion définie par l'utilisateur entre deux types valeur, la même conversion peut également être utilisée entre les types valeur Nullable correspondants.

Boxing et unboxing

Une instance d'un type valeur Nullable `T?` est convertie comme suit :

- Si HasValue retourne `false`, la référence null est générée.
- Si HasValue retourne `true` , la valeur correspondante du type valeur sous-jacent `T` est boxed, et non l'instance de Nullable<T> .

Vous pouvez convertir une valeur boxed d'un type valeur `T` en type valeur Nullable correspondant `T?` , comme le montre l'exemple suivant :

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

Comment identifier un type valeur Nullable

L'exemple suivant montre comment déterminer si une System.Type instance représente un type valeur Nullable construit, autrement dit, le System.Nullable<T> type avec un paramètre de type spécifié `T` :

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

Comme le montre l'exemple, vous utilisez l'opérateur typeof pour créer une System.Type instance.

Si vous souhaitez déterminer si une instance est d'un type valeur Nullable, n'utilisez pas la Object.GetType méthode pour obtenir une Type instance à tester avec le code précédent. Quand vous appelez la Object.GetType méthode sur une instance d'un type valeur Nullable, l'instance est convertie en Object . La conversion boxing d'une instance non null d'un type valeur Nullable équivaut au boxing d'une valeur du type sous-jacent, GetType retourne une Type instance qui représente le type sous-jacent d'un type valeur Nullable :

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

En outre, n'utilisez pas l'opérateur is pour déterminer si une instance est d'un type valeur Nullable. Comme le montre l'exemple suivant, vous ne pouvez pas distinguer les types d'une instance de type valeur Nullable et son instance de type sous-jacent avec l' `is` opérateur :

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Vous pouvez utiliser le code présenté dans l'exemple suivant pour déterminer si une instance est d'un type valeur Nullable :

```
int? a = 14;
Console.WriteLine(IsOfNullableType(a));  // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b));  // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}
```

> **NOTE**
>
> Les méthodes décrites dans cette section ne sont pas applicables dans le cas des types référence Nullable.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types Nullable
- Opérateurs levés
- Conversions Nullable implicites
- Conversions Nullable explicites
- Opérateurs de conversion levés

Voir aussi

- Informations de référence sur C#
- Que signifie réellement « lifted » ?
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- Types références Nullables

Types référence (informations de référence sur C#)

02/11/2020 • 2 minutes to read • Edit Online

Il existe deux genres de types en C# : les types référence et les types valeur. Les variables des types référence font référence à leurs données (objets), tandis que les variables des types valeur contiennent directement leurs données. Avec les types référence, deux variables peuvent faire référence au même objet ; par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, chaque variable a sa propre copie des données et les opérations sur une variable ne peuvent pas affecter l'autre (sauf pour les variables de paramètre in, ref et out ; consultez les modificateurs de paramètre in, ref et out).

Les mots clés suivants sont utilisés pour déclarer des types référence :

- class
- interface
- delegate

C# fournit également les types référence intégrés suivants :

- dynamic
- object
- string

Voir aussi

- Référence C#
- Mots clés C#
- Types de pointeur
- Types de valeur

Types référence intégrés (Référence C#)

02/11/2020 • 12 minutes to read • Edit Online

C# a un nombre de types référence intégrés. Ils ont des mots clés ou des opérateurs qui sont synonymes pour un type dans la bibliothèque .NET.

Type d'objet.

Le type `object` est un alias de System.Object dans .NET. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type System.Object. Vous pouvez assigner des valeurs de tout type aux variables de type `object`. Toute variable `object` peut être attribuée à sa valeur par défaut à l'aide du littéral `null`. Quand une variable d'un type valeur est convertie en type objet, elle est dite *boxed*. Quand une variable de type `object` est convertie en un type valeur, elle est dite *unboxed*. Pour plus d'informations, consultez Conversion boxing et unboxing.

Type de chaîne

Le type `string` représente une séquence de zéro, un ou plusieurs caractères Unicode. `string` est un alias de System.String dans .NET.

Bien que `string` soit un type référence, les opérateurs d'égalité `==` (`!=` et) sont définis pour comparer les valeurs des objets `string`, pas les références. Cela permet de tester l'égalité de chaînes de façon plus intuitive. Par exemple :

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

Cette opération affiche « True », puis « False », car le contenu des chaînes est équivalent, mais `a` et `b` ne font pas référence à la même instance de chaîne.

L'opérateur + concatène les chaînes :

```
string a = "good " + "morning";
```

Cela crée un objet String qui contient « good morning ».

Les chaînes sont *immuables* : il est impossible de changer le contenu d'un objet String après avoir créé l'objet, bien que la syntaxe semble indiquer le contraire. Par exemple, lorsque vous écrivez ce code, le compilateur crée en fait un nouvel objet String pour stocker la nouvelle séquence de caractères, et ce nouvel objet est attribué à `b`. La mémoire allouée pour `b` (quand elle contenait la chaîne « h ») est alors éligible pour le garbage collection.

```
string b = "h";
b += "ello";
```

L' `[]` opérateur peut être utilisé pour l'accès en lecture seule aux caractères individuels d'une chaîne. Les valeurs d'index valides commencent à `0` et doivent être inférieures à la longueur de la chaîne :

```
string str = "test";
char x = str[2];  // x = 's';
```

De la même façon, l' `[]` opérateur peut également être utilisé pour effectuer une itération au sein de chaque caractère dans une chaîne :

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

Les littéraux de chaîne sont de type `string` et peuvent être écrits sous deux formes, entre guillemets et entre `@`. Les littéraux de chaîne entre guillemets sont placés entre guillemets doubles (") :

```
"good morning"  // a string literal
```

Les littéraux de chaîne peuvent contenir tout littéral de caractère. Les séquences d'échappement sont incluses. L'exemple suivant utilise la séquence d'échappement `\\` pour la barre oblique inverse, `\u0066` pour la lettre f et `\n` pour un saut de ligne.

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

NOTE

Le code d'échappement `\udddd` (où `dddd` est un nombre à quatre chiffres) représente le caractère Unicode U+`dddd`. Les codes d'échappement Unicode à huit chiffres sont également reconnus : `\Udddddddd`.

Les littéraux de chaîne textuelle commencent par `@` et sont placés entre guillemets doubles. Par exemple :

```
@"good morning"  // a string literal
```

L'avantage des chaînes textuelles est que les séquences d'échappement *ne sont pas* traitées, ce qui facilite l'écriture, par exemple, d'un nom de fichier complet Windows :

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

Pour inclure un guillemet double dans une chaîne @-quoted, doublez-le :

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Type de délégué

La déclaration d'un type délégué est semblable à une signature de méthode. Elle a une valeur de retour et un nombre quelconque de paramètres de type quelconque :

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

Dans .NET, les types `System.Action` et `System.Func` fournissent des définitions génériques pour de nombreux délégués courants. Vous n'avez probablement pas besoin de définir de nouveaux types de délégué personnalisés. À la place, vous pouvez créer des instanciations des types génériques fournis.

Un `delegate` est un type référence qui peut être utilisé pour encapsuler une méthode anonyme ou nommée. Les délégués sont comparables aux pointeurs fonction en C++, mais ils offrent l'avantage d'être sûrs et de type sécurisé. Pour les applications de délégués, consultez Délégués et Délégués génériques. Les délégués sont la base des événements. Un délégué peut être instancié en l'associant à une méthode nommée ou anonyme.

Le délégué doit être instancié avec une méthode ou une expression lambda qui a un type de retour compatible et des paramètres d'entrée. Pour plus d'informations sur le degré de variance autorisé dans la signature de méthode, consultez Variance dans les délégués. Pour une utilisation avec des méthodes anonymes, le délégué et le code à lui associer sont déclarés ensemble.

Type dynamique

Le type `dynamic` indique que l'utilisation de la variable et des références à ses membres contourne la vérification du type au moment de la compilation. Au lieu de cela, ces opérations sont résolues au moment de l'exécution. Le type `dynamic` simplifie l'accès aux API COM telles que les API Office Automation, aux API dynamiques telles que les bibliothèques IronPython et au modèle DOM (Document Object Model) HTML.

Le type `dynamic` se comporte comme le type `object` dans la plupart des cas. En particulier, toute expression non null peut être convertie en type `dynamic`. Le type `dynamic` diffère de `object` en cela que les opérations qui contiennent des expressions de type `dynamic` ne sont pas résolues et leur type n'est pas vérifié par le compilateur. Le compilateur empaquète des informations sur l'opération, qui sont ensuite utilisées pour évaluer l'opération au moment de l'exécution. Dans le cadre du processus, les variables de type `dynamic` sont compilées dans des variables de type `object`. Ainsi, le type `dynamic` existe seulement au moment de la compilation, et non au moment de l'exécution.

L'exemple suivant compare une variable de type `dynamic` à une variable de type `object`. Pour vérifier le type de chaque variable au moment de la compilation, placez le pointeur de la souris sur `dyn` ou `obj` dans les instructions `WriteLine`. Copiez le code suivant dans un éditeur où IntelliSense est disponible. IntelliSense affiche **dynamic** pour `dyn` et **object** pour `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

Les instructions WriteLine affichent les types d'exécution de `dyn` et `obj`. À ce stade, tous deux ont le même type, entier. La sortie suivante est produite :

```
System.Int32
System.Int32
```

Pour voir la différence entre `dyn` et `obj` au moment de la compilation, ajoutez les deux lignes suivantes entre les déclarations et les instructions `WriteLine` de l'exemple précédent.

```
dyn = dyn + 3;
obj = obj + 3;
```

Une erreur du compilateur est signalée pour la tentative d'ajout d'un entier et un d'objet dans l'expression `obj + 3`. Toutefois, aucune erreur n'est signalée pour `dyn + 3`. L'expression qui contient `dyn` n'est pas vérifié au moment de la compilation, car le type de `dyn` est `dynamic`.

L'exemple suivant utilise `dynamic` dans plusieurs déclarations. La méthode `Main` compare également la vérification de type au moment de la compilation avec la vérification de type au moment de l'exécution.

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

Voir aussi

- Référence C#
- Mots clés C#
- Événements
- Utilisation du type dynamic
- Meilleures pratiques pour l'utilisation de chaînes
- Opérations de chaînes de base
- Création de chaînes
- Opérateurs de conversion et de test de type
- Comment effectuer un cast en toute sécurité à l'aide de critères spéciaux et des opérateurs As

et is

- Procédure pas à pas : création et utilisation d'objets dynamiques
- System.Object
- System.String
- System.Dynamic.DynamicObject

class (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Les classes sont déclarées à l'aide du mot clé `class`, comme l'illustre l'exemple suivant :

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Remarques

Le langage C# ne permet qu'un seul héritage. Cela signifie qu'une classe peut uniquement hériter de l'implémentation d'une seule classe de base. Toutefois, une classe peut implémenter plusieurs interfaces. Le tableau suivant répertorie des exemples d'héritage de classe et d'implémentation d'interface :

HÉRITAGE	EXEMPLE
Aucune	`class ClassA { }`
Unique	`class DerivedClass : BaseClass { }`
Aucun, implémente deux interfaces	`class ImplClass : IFace1, IFace2 { }`
Unique, implémente une seule interface	`class ImplDerivedClass : BaseClass, IFace1 { }`

Les classes que vous déclarez directement dans un espace de noms, non imbriquées dans d'autres classes, peuvent être public ou internal. Par défaut, les classes sont `internal`.

Les membres de classe, notamment les classes imbriquées, peuvent être public, protected internal, protected, internal, private ou private protected. Par défaut, ils sont `private`.

Pour plus d'informations, consultez Modificateurs d'accès.

Vous pouvez déclarer des classes génériques qui ont des paramètres de type. Pour plus d'informations, consultez Classes génériques.

Une classe peut contenir les déclarations des membres suivants :

- Constructeurs
- Constantes
- Fields
- Finaliseurs
- Méthodes
- Propriétés

- Indexeurs
- Opérateurs
- Événements
- Délégués
- Classes
- Interfaces
- Types de structure
- Types d'énumération

Exemple

L'exemple suivant explique comment déclarer des champs, des constructeurs et des méthodes de classe. Il illustre également l'instanciation d'un objet et l'impression des données d'une instance. Dans cet exemple, deux classes sont déclarées. La première, `Child`, contient deux champs privés (`name` et `age`), deux constructeurs publics et une méthode publique. La deuxième, `StringTest`, contient `Main`.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Commentaires

Notez que, dans l'exemple précédant, les champs privés (`name` et `age`) ne sont accessibles que par le biais de la méthode publique de la classe `Child`. Par exemple, vous ne pouvez pas imprimer le nom de l'enfant à partir de la méthode `Main` en utilisant une instruction comme celle-ci :

```
Console.Write(child1.name);   // Error
```

L'accès aux membres privés de `Child` à partir de `Main` est uniquement possible si `Main` est un membre de la classe.

Du fait que les types déclarés dans une classe sans modificateur d'accès sont par défaut `private`, les données membres dans cet exemple sont toujours `private` si le mot clé est supprimé.

Notez enfin que pour l'objet créé à l'aide du constructeur sans paramètre (`child3`), le champ `age` est initialisé par défaut à la valeur zéro.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Types référence

interface (Référence C#)

16/12/2020 • 5 minutes to read • Edit Online

Une interface définit un contrat. Tout `class` ou `struct` qui implémente ce contrat doit fournir une implémentation des membres définis dans l'interface. À compter de C# 8,0, une interface peut définir une implémentation par défaut pour les membres. Il peut également définir des membres afin de `static` fournir une implémentation unique pour les fonctionnalités communes.

Dans l'exemple suivant, la classe `ImplementationClass` doit implémenter une méthode nommée `SampleMethod` qui n'a aucun paramètre et qui retourne `void`.

Pour plus d'informations et d'exemples, consultez Interfaces.

Exemple

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Une interface peut être membre d'un espace de noms ou d'une classe. Une déclaration d'interface peut contenir des déclarations (signatures sans implémentation) des membres suivants :

- Méthodes
- Propriétés
- Indexeurs
- Événements

Les déclarations de membre précédentes ne contiennent généralement pas de corps. À compter de C# 8,0, un membre d'interface peut déclarer un corps. C'est ce qu'on appelle une *implémentation par défaut*. Les membres avec des corps permettent à l'interface de fournir une implémentation « par défaut » pour les classes et les structs qui ne fournissent pas d'implémentation de substitution. En outre, à compter de C# 8,0, une interface peut inclure les éléments suivants :

- Constantes
- Opérateurs

- Constructeur statique.
- Types imbriqués
- Champs, méthodes, propriétés, indexeurs et événements statiques
- Déclarations de membre à l'aide de la syntaxe d'implémentation d'interface explicite.
- Modificateurs d'accès explicites (l'accès par défaut est `public`).

Les interfaces ne peuvent pas contenir l'état de l'instance. Alors que les champs statiques sont désormais autorisés, les champs d'instance ne sont pas autorisés dans les interfaces. Les propriétés auto des instances ne sont pas prises en charge dans les interfaces, car elles déclarent implicitement un champ masqué. Cette règle a un effet subtil sur les déclarations de propriété. Dans une déclaration d'interface, le code suivant ne déclare pas une propriété implémentée automatiquement comme elle le fait dans un `class` ou un `struct` . Au lieu de cela, elle déclare une propriété qui n'a pas d'implémentation par défaut, mais qui doit être implémentée dans tout type qui implémente l'interface :

```
public interface INamed
{
  public string Name {get; set;}
}
```

Une interface peut hériter d'une ou de plusieurs interfaces de base. Quand une interface se substitue à une méthode implémentée dans une interface de base, elle doit utiliser la syntaxe d' implémentation d'interface explicite .

Lorsqu'une liste de types de base contient une classe de base et des interfaces, la classe de base doit figurer en premier dans la liste.

Une classe qui implémente une interface peut implémenter explicitement les membres de cette interface. Un membre implémenté explicitement n'est pas accessible via une instance de classe, mais uniquement via une instance de l'interface. En outre, les membres d'interface par défaut sont accessibles uniquement par le biais d'une instance de l'interface.

Pour plus d'informations sur l'implémentation d'interface explicite, consultez implémentation d'interface explicite.

Exemple

L'exemple suivant montre une implémentation d'interface. Dans cet exemple, l'interface contient la déclaration de propriété, et la classe contient l'implémentation. Toutes les instances d'une classe qui implémentent `IPoint` ont les propriétés entières `x` et `y`.

```
interface IPoint
{
   // Property signatures:
   int X
   {
      get;
      set;
   }

   int Y
   {
      get;
      set;
   }

   double Distance
   {
       get;
   }
}

class Point : IPoint
{
   // Constructor:
   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }

   // Property implementation:
   public int X { get; set; }

   public int Y { get; set; }

   // Property implementation
   public double Distance =>
      Math.Sqrt(X * X + Y * Y);

}

class MainClass
{
   static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.X, p.Y);
   }

   static void Main()
   {
      IPoint p = new Point(2, 3);
      Console.Write("My Point: ");
      PrintPoint(p);
   }
}
// Output: My Point: x=2, y=3
```

spécification du langage C#

Pour plus d'informations, consultez la section interfaces de la spécification du langage C# et la spécification des fonctionnalités pour les membres d'interface par défaut-C# 8,0

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Types référence
- Interfaces
- Utilisation de propriétés
- Utilisation d'indexeurs

Types référence Nullable (référence C#)

16/12/2020 • 10 minutes to read • Edit Online

> **NOTE**
>
> Cet article traite des types de référence Nullable. Vous pouvez également déclarer des types valeur Nullable.

Les types référence Nullable sont disponibles à partir de C# 8,0, dans le code qui a opté pour un *contexte reconnaissant* la valeur null. Les types de référence Nullable, les avertissements d'analyse statique null et l' opérateur null-indulgent avec sont des fonctionnalités de langage facultatives. Tous sont désactivés par défaut. Un *contexte Nullable* est contrôlé au niveau du projet à l'aide des paramètres de génération, ou dans le code à l'aide de pragmas.

Dans un contexte qui prend en charge la valeur NULL :

- Une variable d'un type référence `T` doit être initialisée avec une valeur non null et ne peut jamais être assignée à une valeur qui peut être `null` .
- Une variable d'un type référence `T?` peut être initialisée avec `null` ou assignée `null` , mais doit être vérifiée `null` avant d'être déréférencée.
- Une variable `m` de type `T?` est considérée comme non NULL lorsque vous appliquez l'opérateur null-indulgent avec, comme dans `m!` .

Les différences entre un type de référence non Nullable `T` et un type de référence Nullable `T?` sont appliquées par l'interprétation du compilateur des règles précédentes. Une variable de type `T` et une variable de type `T?` sont représentées par le même type .net. L'exemple suivant déclare une chaîne non Nullable et une chaîne Nullable, puis utilise l'opérateur null-indulgent avec pour assigner une valeur à une chaîne qui n'accepte pas les valeurs NULL :

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

Les variables `notNull` et `nullable` sont toutes les deux représentées par le String type. Étant donné que les types non Nullable et Nullable sont tous deux stockés sous le même type, il existe plusieurs emplacements où l'utilisation d'un type de référence Nullable n'est pas autorisée. En général, un type de référence Nullable ne peut pas être utilisé comme classe de base ou comme interface implémentée. Un type de référence Nullable ne peut pas être utilisé dans une expression de création d'objet ou de test de type. Un type de référence Nullable ne peut pas être le type d'une expression d'accès de membre. Les exemples suivants illustrent ces constructions :

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Références Nullable et analyse statique

Les exemples de la section précédente illustrent la nature des types de référence Nullable. Les types de référence Nullable ne sont pas de nouveaux types de classe, mais plutôt des annotations sur les types de référence existants. Le compilateur utilise ces annotations pour vous aider à identifier les erreurs de référence null potentielles dans votre code. Il n'existe aucune différence d'exécution entre un type de référence non Nullable et un type de référence Nullable. Le compilateur n'ajoute pas de vérification du runtime pour les types de référence non Nullable. Les avantages sont liés à l'analyse au moment de la compilation. Le compilateur génère des avertissements qui vous aident à trouver et à corriger les erreurs null potentielles dans votre code. Vous déclarez votre intention et le compilateur vous avertit lorsque votre code viole cet objectif.

Dans un contexte activé avec la valeur null, le compilateur effectue une analyse statique sur les variables de tout type référence, à la fois Nullable et non Nullable. Le compilateur effectue le suivi de l'État null de chaque variable de référence comme *not null* ou *peut être null*. L'État par défaut d'une référence qui n'accepte pas les valeurs NULL n'est *pas null*. L'État par défaut d'une référence Nullable est *peut-être null*.

Les types référence non Nullable doivent toujours être déréférencés, car leur état NULL n'est *pas null*. Pour appliquer cette règle, le compilateur émet des avertissements si un type de référence non Nullable n'est pas initialisé à une valeur non null. Les variables locales doivent être assignées là où elles sont déclarées. Chaque constructeur doit assigner chaque champ, dans son corps, à un constructeur appelé ou à l'aide d'un initialiseur de champ. Le compilateur émet des avertissements si une référence qui n'accepte pas les valeurs NULL est assignée à une référence dont l'État est *peut-être null*. Toutefois, étant donné qu'une référence non Nullable n'est *pas null*, aucun avertissement n'est émis lorsque ces variables sont déréférencées.

Les types référence Nullable peuvent être initialisés ou assignés à `null` . Par conséquent, l'analyse statique doit déterminer qu'une variable n'a *pas la valeur null* avant d'être déréférencée. Si une référence Nullable est peut-être *null*, elle ne peut pas être assignée à une variable de référence n'acceptant pas les valeurs NULL. La classe suivante montre des exemples de ces avertissements :

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! short description not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

L'extrait de code suivant indique où le compilateur émet des avertissements lors de l'utilisation de cette classe :

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows shortDescription
maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

Les exemples précédents illustrent l'analyse statique du compilateur pour déterminer l'État null des variables de référence. Le compilateur applique des règles de langage pour les vérifications et les assignations null afin d'en informer l'analyse. Le compilateur ne peut pas faire d'hypothèses sur la sémantique des méthodes ou des propriétés. Si vous appelez des méthodes qui effectuent des vérifications de valeur null, le compilateur ne peut pas savoir que ces méthodes affectent l'État null d'une variable. Vous pouvez ajouter plusieurs attributs à vos API pour informer le compilateur de la sémantique des arguments et des valeurs de retour. Ces attributs ont été appliqués à de nombreuses API communes dans les bibliothèques .NET Core. Par exemple, IsNullOrEmpty a été

mis à jour et le compilateur interprète correctement cette méthode comme une vérification de valeur null. Pour plus d'informations sur les attributs qui s'appliquent à l'analyse statique d'État null, consultez l'article sur les attributs Nullable.

Définition du contexte Nullable

Il existe deux façons de contrôler le contexte Nullable. Au niveau du projet, vous pouvez ajouter le `<Nullable>enable</Nullable>` paramètre de projet. Dans un seul fichier source C#, vous pouvez ajouter le `#nullable enable` pragma pour activer le contexte Nullable. Consultez l'article sur la définition d'une stratégie Nullable.

spécification du langage C#

Pour plus d'informations, consultez les propositions suivantes pour la spécification du langage C#:

- Types références Nullables
- Spécification des types de référence Nullable Draft

Voir aussi

- Informations de référence sur C#
- Types valeur Nullable

void (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Vous utilisez `void` comme type de retour d'une méthode (ou une fonction locale) pour spécifier que la méthode ne retourne pas de valeur.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

Vous pouvez également utiliser `void` comme type référent pour déclarer un pointeur vers un type inconnu. Pour plus d'informations, consultez Types pointeur.

Vous ne pouvez pas utiliser `void` comme type de variable.

Voir aussi

- Informations de référence sur C#
- System.Void

var (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

À compter de C# 3, les variables déclarées au niveau de la portée de la méthode peuvent avoir un « type » implicite `var` . Une variable locale implicitement typée est fortement typée comme si vous aviez déclaré vous-même le type, alors que c'est le compilateur qui détermine le type. Les deux déclarations suivantes d' `i` sont équivalentes fonctionnellement :

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

IMPORTANT

Quand `var` est utilisé avec les types référence Nullable activés, il implique toujours un type référence Nullable même si le type d'expression n'accepte pas la valeur null.

Le `var` mot clé est couramment utilisé avec les expressions d'appel de constructeur. L'utilisation de `var` vous permet de ne pas répéter un nom de type dans une déclaration de variable et une instanciation d'objet, comme le montre l'exemple suivant :

```
var xs = new List<int>();
```

À compter de C# 9,0, vous pouvez utiliser une `new` expression de type cible comme alternative :

```
List<int> xs = new();
List<int>? ys = new();
```

Exemple

L'exemple suivant présente deux expressions de requête. Dans la première expression, l'utilisation de `var` est autorisée, mais n'est pas obligatoire parce que le type du résultat de la requête peut être déclaré explicitement en tant que `IEnumerable<string>`. Toutefois, dans la deuxième expression où `var` est utilisé, le résultat peut être une collection de types anonymes, et le nom de ce type n'est pas accessible sauf par le compilateur. Si vous utilisez `var`, vous n'avez pas besoin de créer une classe pour le résultat. Notez que, dans le deuxième exemple, la variable d'itération `foreach``item` doit également être implicitement typée.

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

Voir aussi

- Informations de référence sur C#
- Variables locales implicitement typées
- Relations de types dans les opérations de requête LINQ

Types intégrés (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le tableau suivant répertorie les types valeur intégrés C# :

MOT CLÉ C# TYPE	TYPE .NET
`bool`	System.Boolean
`byte`	System.Byte
`sbyte`	System.SByte
`char`	System.Char
`decimal`	System.Decimal
`double`	System.Double
`float`	System.Single
`int`	System.Int32
`uint`	System.UInt32
`long`	System.Int64
`ulong`	System.UInt64
`short`	System.Int16
`ushort`	System.UInt16

Le tableau suivant répertorie les types de référence intégrés C# :

MOT CLÉ C# TYPE	TYPE .NET
`object`	System.Object
`string`	System.String
`dynamic`	System.Object

Dans les tableaux précédents, chaque mot clé de type C# de la colonne de gauche est un alias pour le type .NET correspondant. Ils sont interchangeables. Par exemple, les déclarations suivantes déclarent des variables du même type :

```
int a = 123;
System.Int32 b = 123;
```

Le `void` mot clé représente l'absence d'un type. Vous l'utilisez comme type de retour d'une méthode qui ne retourne pas de valeur.

Voir aussi

- Informations de référence sur C#
- Valeurs par défaut des types C#

Types non managés (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Un type est un **type non managé** s'il s'agit de l'un des types suivants :

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` ou `bool`
- Tout type enum
- Tout type pointeur
- Tout type struct défini par l'utilisateur qui contient des champs de types non managés uniquement et, en C# 7,3 et antérieur, n'est pas un type construit (un type qui inclut au moins un argument de type)

À compter de C# 7,3, vous pouvez utiliser la `unmanaged` contrainte pour spécifier qu'un paramètre de type est un type non managé qui n'est pas un pointeur, non Nullable.

À compter de C# 8,0, un type struct *construit* qui contient des champs de types non managés uniquement est également non managé, comme le montre l'exemple suivant :

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

Un struct générique peut être la source des types construits non managés et non managés. L'exemple précédent définit un struct générique `Coords<T>` et présente les exemples de types construits non managés. L'exemple de n'est pas un type non managé `Coords<object>` . Elle n'est pas non plus gérée, car elle contient les champs du `object` type, qui ne sont pas non managés. Si vous souhaitez que *tous les* types construits soient des types non managés, utilisez la `unmanaged` contrainte dans la définition d'un struct générique :

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

spécification du langage C#

Pour plus d'informations, consultez la section Types de pointeur de la Spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de pointeur
- Types liés à la mémoire et à l'étendue
- Opérateur sizeof
- stackalloc

Valeurs par défaut des types C (référence C)

27/03/2020 • 2 minutes to read • Edit Online

Le tableau suivant présente les valeurs par défaut des types C# :

TYPE	VALEUR PAR DÉFAUT
tout type référence ;	`null`
Tout type numérique intégral intégré	0 (zéro)
Tout type numérique à virgule flottante intégré	0 (zéro)
Bool	`false`
char	`'\0'` (U+0000)
Enum	Valeur produite par l'expression `(E)0`, où `E` est l'identificateur de l'enum.
struct	Valeur produite en affectant à tous les champs de type valeur leur valeur par défaut et à tous les champs de type référence la valeur `null`.
Tout type valeur Nullable	Instance pour laquelle la propriété HasValue a la valeur `false` et la propriété Value n'est pas définie. Cette valeur par défaut est également connue sous le nom de valeur *nulle* d'un type de valeur nulle.

Utilisez `default` l'opérateur pour produire la valeur par défaut d'un type, comme le montre l'exemple suivant :

```
int a = default(int);
```

En commençant par le C 7.1, vous pouvez utiliser le `default` littéral pour initialiser une variable avec la valeur par défaut de son type :

```
int a = default;
```

Pour un type valeur, le constructeur sans paramètre implicite produit également la valeur par défaut du type, comme le montre l'exemple suivant :

```
var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

Au moment de System.Type l'exécution, si l'instance Activator.CreateInstance(Type) représente un type de valeur, vous pouvez utiliser la méthode pour invoquer le constructeur sans paramètres pour obtenir la valeur par défaut du type.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Valeurs par défaut
- Constructeurs par défaut

Voir aussi

- Référence C#
- Constructeurs

Mots clés C#

16/12/2020 • 3 minutes to read • Edit Online

Les mots clés sont des identificateurs réservés prédéfinis, qui ont des significations spécifiques pour le compilateur. Ils ne peuvent pas être utilisés comme identificateurs dans votre programme, sauf s'ils incluent `@` comme préfixe. Par exemple, `@if` est un identificateur valide, mais pas `if`, car `if` est un mot clé.

Le premier tableau de cette rubrique répertorie les mots clés qui sont des identificateurs réservés quelle que soit la partie d'un programme C#. Le deuxième tableau de cette rubrique liste les mots clés contextuels en C#. Les mots clés contextuels ont une signification spéciale dans un contexte de programme limité ; ils peuvent être utilisés comme identificateurs en dehors de ce contexte. En règle générale, lorsque de nouveaux mots clés sont ajoutés au langage C#, ils sont ajoutés comme mots clés contextuels afin d'éviter l'interruption des programmes écrits dans les versions antérieures.

abstract	en tant que	base	bool	
break	byte	case	catch	
char	désactivé	class	const	
continue	decimal	default	delegate	
do	double	else	variables	
event	explicitement	extern	false	
suivie	des	float	for	
foreach	goto	if	Implicit	
in	int	interface	internal	
is	lock	long	namespace	
nouveau	null	object	operator	
à	override	params	priv	
protected	public	seulement	ref	
return	sbyte	sealed	short	
sizeof	stackalloc	static	string	
modélis	switch	this	throw	
true	Essayez	typeof	uint	

ulong	unchecked	unsafe	ushort	
à	virtual	void	volatile	
while				

Mots clés contextuels

Un mot clé contextuel sert à donner une signification spécifique dans le code, sans pour autant être un mot réservé en C#. Certains mots clés contextuels, tels que `partial` et `where`, ont des significations spéciales dans deux contextes ou plus.

add	alias	ascending
async	await	by
descending	dynamic	equals
from	get	Généralités
groupe	into	join
let	nameof	NOTNULL
actif	TriPar	Partial (type)
Partial (méthode)	remove	select
set	non managé (contrainte de type générique)	value
var	when (condition de filtre)	where (contrainte de type générique)
where (clause de requête)	with	yield

Voir aussi

- Informations de référence sur C#

Modificateurs d'accès (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Les modificateurs d'accès sont des mots clés utilisés pour spécifier l'accessibilité déclarée d'un membre ou d'un type. Cette section présente les quatre modificateurs d'accès :

- `public`
- `protected`
- `internal`
- `private`

Les modificateurs d'accès permettent de spécifier les six niveaux d'accessibilité suivants :

- `public` : L'accès n'est pas limité.
- `protected` : L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur.
- `internal` : L'accès est limité à l'assembly actuel.
- `protected internal` : L'accès est limité à l'assembly actuel ou aux types dérivés de la classe conteneur.
- `private` : L'accès est limité au type conteneur.
- `private protected` : L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur dans l'assembly actuel.

Cette section introduit également les éléments suivants :

- Niveaux d'accessibilité : utilisation des quatre modificateurs d'accès pour déclarer six niveaux d'accessibilité.
- Domaine d'accessibilité : spécifie où, dans les sections du programme, un membre peut être référencé.
- Limitations sur l'utilisation des niveaux d'accessibilité : résumé des restrictions sur l'utilisation des niveaux d'accessibilité déclarés.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Mots clés d'accès
- Modificateurs

Niveaux d'accessibilité (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Utilisez les modificateurs d'accès `public`, `protected`, `internal` ou `private` pour spécifier l'un des niveaux d'accessibilité déclarés ci-dessous pour les membres.

ACCESSIBILITÉ DÉCLARÉE	SIGNIFICATION
`public`	L'accès n'est pas limité.
`protected`	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur.
`internal`	L'accès est limité à l'assembly actuel.
`protected internal`	L'accès est limité à l'assembly actuel ou aux types dérivés de la classe conteneur.
`private`	L'accès est limité au type conteneur.
`private protected`	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur dans l'assembly actuel. Disponible depuis C# 7.2.

Vous ne pouvez spécifier qu'un seul modificateur d'accès pour un membre ou un type, sauf si vous utilisez les combinaisons `protected internal` ou `private protected`.

Les modificateurs d'accès ne sont pas autorisés sur les espaces de noms. Les espaces de noms ne présentent aucune limitation d'accès.

Selon le contexte dans lequel une déclaration de membre est effectuée, seules certaines accessibilités déclarées sont autorisées. Si aucun modificateur d'accès n'est spécifié dans une déclaration de membre, une accessibilité par défaut est utilisée.

Les types de premier niveau, qui ne sont pas imbriqués dans d'autres types, peuvent uniquement avoir une accessibilité `internal` ou `public`. L'accessibilité par défaut de ces types est `internal`.

Les types imbriqués, qui sont membres d'autres types, peuvent avoir les accessibilités déclarées indiquées dans le tableau suivant.

MEMBRES DE	ACCESSIBILITÉ PAR DÉFAUT DU MEMBRE	ACCESSIBILITÉ DÉCLARÉE DU MEMBRE AUTORISÉE
`enum`	`public`	None

MEMBRES DE	ACCESSIBILITÉ PAR DÉFAUT DU MEMBRE	ACCESSIBILITÉ DÉCLARÉE DU MEMBRE AUTORISÉE

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	None

`struct`	`private`	`public`
		`internal`
		`private`

L'accessibilité d'un type imbriqué dépend de son domaine d'accessibilité, qui est déterminé à la fois par l'accessibilité déclarée du membre et par le domaine d'accessibilité du type conteneur immédiat. Toutefois, le domaine d'accessibilité d'un type imbriqué ne peut pas dépasser celui du type conteneur.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Domaine d'accessibilité
- Restrictions concernant l'utilisation des niveaux d'accessibilité
- Modificateurs d'accès
- public
- priv
- protected
- internal

Domaine d'accessibilité (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le domaine d'accessibilité d'un membre indique dans quelles sections du programme un membre peut être référencé. Si le membre est imbriqué dans un autre type, son domaine d'accessibilité est déterminé à la fois par le niveau d'accessibilité du membre et par le domaine d'accessibilité du type conteneur immédiat.

Le domaine d'accessibilité d'un type de niveau supérieur est au moins le texte du programme du projet dans lequel il est déclaré. Cela signifie que le domaine inclut tous les fichiers sources de ce projet. Le domaine d'accessibilité d'un type imbriqué est au moins le texte de programme du type dans lequel il est déclaré. Autrement dit, le domaine est le corps du type, qui inclut tous les types imbriqués. Le domaine d'accessibilité d'un type imbriqué ne dépasse jamais celui du type conteneur. Ces concepts sont illustrés dans l'exemple suivant.

Exemple

Cet exemple comporte un type de niveau supérieur, `T1`, et deux classes imbriquées, `M1` et `M2`. Les classes contiennent des champs qui ont des accessibilités déclarées différentes. Dans la méthode `Main`, un commentaire suit chaque instruction pour indiquer le domaine d'accessibilité de chaque membre. Notez que les instructions qui essaient de référencer les membres inaccessibles sont commentées. Si vous souhaitez voir les erreurs de compilation provoquées par le référencement d'un membre inaccessible, supprimez les commentaires un par un.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
```

```
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Restrictions concernant l'utilisation des niveaux d'accessibilité
- Modificateurs d'accès
- public
- priv
- protected
- internal

Limitations sur l'utilisation des niveaux d'accessibilité (informations de référence sur C#)

02/11/2020 • 4 minutes to read • Edit Online

Lorsque vous spécifiez un type dans une déclaration, vérifiez si le niveau d'accessibilité du type dépend du niveau d'accessibilité d'un membre ou d'un autre type. Par exemple, la classe de base directe doit être au moins aussi accessible que la classe dérivée. Les déclarations suivantes entraînent une erreur du compilateur, car la classe de base `BaseClass` est moins accessible que `MyClass` :

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

Le tableau suivant résume les limitations sur les niveaux d'accessibilité déclarés.

CONTEXT	REMARQUES
Classes	La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même.
Interfaces	Les interfaces de base explicites d'un type d'interface doivent être au moins aussi accessibles que le type d'interface lui-même.
Délégués	Le type de retour et les types de paramètres d'un type délégué doivent être au moins aussi accessibles que le type délégué lui-même.
Constantes	Le type d'une constante doit être au moins aussi accessible que la constante elle-même.
Fields	Le type d'un champ doit être au moins aussi accessible que le champ lui-même.
Méthodes	Le type de retour et les types de paramètres d'une méthode doivent être au moins aussi accessibles que la méthode elle-même.
Propriétés	Le type d'une propriété doit être au moins aussi accessible que la propriété elle-même.
Événements	Le type d'un événement doit être au moins aussi accessible que l'événement lui-même.
Indexeurs	Le type et les types de paramètres d'un indexeur doivent être au moins aussi accessibles que l'indexeur lui-même.
Opérateurs	Le type de retour et les types de paramètres d'un opérateur doivent être au moins aussi accessibles que l'opérateur lui-même.

CONTEXT	REMARQUES
Constructeurs	Les types de paramètres d'un constructeur doivent être au moins aussi accessibles que le constructeur lui-même.

Exemple

L'exemple suivant contient des déclarations erronées de différents types. Le commentaire qui suit chaque déclaration indique l'erreur du compilateur à attendre.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Domaine d’accessibilité
- Niveaux d'accessibilité
- Modificateurs d’accès
- public
- priv
- protected
- intérieurs

internal (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le mot clé `internal` est un modificateur d'accès pour les types et les membres de type.

> Cette page traite de l'accès `internal`. Le `internal` mot clé fait également partie du `protected internal` modificateur d'accès.

Les types et les membres internes (internal) sont accessibles uniquement dans les fichiers d'un même assembly, comme dans l'exemple suivant :

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

Pour obtenir une comparaison de `internal` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité et Modificateurs d'accès.

Pour plus d'informations sur les assemblys, consultez Aassemblys dans .NET.

L'accès interne est fréquemment utilisé lors du développement basé sur les composants, car il permet à un groupe de composants de collaborer de façon privée sans être exposés au reste du code de l'application. Par exemple, un framework de création d'interfaces graphiques utilisateur peut fournir les classes `Control` et `Form` qui coopèrent en utilisant des membres ayant un accès interne. Étant donné que ces membres sont internes, ils ne sont pas exposés au code qui utilise le framework.

Le fait de référencer un type ou un membre avec accès interne en dehors de l'assembly dans lequel il a été défini constitue une erreur.

Exemple

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly1_a.cs`. Le premier fichier contient la classe de base interne `BaseClass`. Dans le deuxième fichier, une tentative d'instanciation de `BaseClass` génère une erreur.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

Exemple

Dans cet exemple, utilisez les mêmes fichiers que vous avez utilisés dans l'exemple 1, et remplacez le niveau d'accessibilité `BaseClass` par `public`. Remplacez également le niveau d'accessibilité du membre `intM` par `internal`. Dans ce cas, vous pouvez instancier la classe, mais vous ne pouvez pas accéder au membre interne.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- priv
- protected

private (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `private` est un modificateur d'accès de membre.

> Cette page traite de l'accès `private`. Le `private` mot clé fait également partie du `private protected` modificateur d'accès.

L'accès privé est le niveau d'accès le moins permissif. Les membres privés sont accessibles uniquement dans le corps de la classe ou le struct dans lequel ils sont déclarés, comme dans cet exemple :

```
class Employee
{
    private int i;
    double d;   // private access by default
}
```

Les types imbriqués dans le même corps peuvent également accéder à ces membres privés.

Référencer un membre privé en dehors d'une classe ou d'un struct où il est déclaré serait à l'origine d'une erreur de compilation.

Pour obtenir une comparaison de `private` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité et Modificateurs d'accès.

Exemple

Dans cet exemple, la classe `Employee` contient deux membres de données privés, `name` et `salary`. S'agissant de membres privés, ils sont accessibles uniquement par les méthodes membres. Les méthodes publiques `GetName` et `Salary` sont ajoutées pour permettre un accès contrôlé aux membres privés. Le membre `name` est accessible via une méthode publique et le membre `salary` est accessible via une propriété publique en lecture seule. (Pour plus d'informations, consultez Propriétés.)

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e.name;
        //    double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- protected
- intérieurs

protected (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

Le mot clé `protected` est un modificateur d'accès de membre.

> **NOTE**
> Cette page traite de l'accès `protected`. Le `protected` mot clé fait également partie des `protected internal` `private protected` modificateurs d'accès et.

Un membre protégé est accessible dans sa classe et par les instances de la classe dérivée.

Pour obtenir une comparaison de `protected` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

Exemple

Un membre protégé d'une classe de base est accessible dans une classe dérivée uniquement si l'accès s'effectue par le biais du type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

L'instruction `a.x = 10` génère une erreur, car elle est appelée dans la méthode statique Main, et pas dans une instance de la classe B.

Les membres de struct ne peuvent pas être protégés, car le struct ne peut pas être hérité.

Exemple

Dans cet exemple, la classe `DerivedPoint` est dérivée de `Point`. Vous pouvez donc accéder aux membres protégés de la classe de base directement à partir de la classe dérivée.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

Si vous changez les niveaux d'accès de `x` et `y` à private, le compilateur affiche les messages d'erreur suivants :

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- internal
- Problèmes de sécurité pour les mots clés virtuels internes

public (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `public` est un modificateur d'accès pour les types et les membres de types. L'accès public est le niveau d'accès le plus permissif. Il n'existe pas de restrictions d'accès aux membres publics, comme dans cet exemple :

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

Pour plus d'informations, consultez Modificateurs d'accès et Niveaux d'accessibilité.

Exemple

Dans l'exemple suivant, deux classes sont déclarées, `PointTest` et `MainClass`. L'accès aux membres publics `x` et `y` de `PointTest` s'effectue directement à partir de `MainClass`.

```
class PointTest
{
    public int x;
    public int y;
}

class MainClass4
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

Si vous remplacez le niveau d'accès `public` par private ou protected, le message d'erreur suivant s'affiche :

'PointTest.y' est inaccessible en raison de son niveau de protection.

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs d'accès

- Mots clés C#
- Modificateurs d’accès
- Niveaux d'accessibilité
- Modificateurs
- priv
- protected
- intérieurs

protected internal (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

La combinaison de mots clés `protected internal` est un modificateur d'accès de membre. Un membre interne protégé est accessible depuis l'assembly actif ou depuis des types dérivés de la classe conteneur. Pour obtenir une comparaison de `protected internal` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

Exemple

Un membre interne protégé d'une classe de base est accessible depuis n'importe quel type au sein de son assembly conteneur. Il est également accessible dans une classe dérivée qui se trouve dans un autre assembly seulement si l'accès s'effectue via une variable du type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly2.cs`. Le premier fichier contient une classe de base publique, `BaseClass`, et une autre classe, `TestAccess`. `BaseClass` possède un membre interne protégé, `myValue`, à laquelle le type `TestAccess` accède. Dans le deuxième fichier, une tentative d'accès à `myValue` via une instance de `BaseClass` génère une erreur, tandis qu'un accès à ce membre via une instance d'une classe dérivée, `DerivedClass`, réussit.

Les membres de struct ne peuvent pas être `protected internal`, car le struct ne peut pas être hérité.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- intérieurs
- Problèmes de sécurité pour les mots clés virtuels internes

private protected (Référence C#)

02/11/2020 • 3 minutes to read • Edit Online

La combinaison de mots clés `private protected` est un modificateur d'accès de membre. Un membre protégé privé est accessible par les types dérivés de la classe conteneur, mais seulement au sein de son assembly conteneur. Pour obtenir une comparaison de `private protected` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

> **NOTE**
>
> Le modificateur d'accès `private protected` est valide dans C# 7.2 et ultérieur.

Exemple

Un membre protégé privé d'une classe de base est accessible à partir des types dérivés de son assembly conteneur seulement si le type statique de la variable est le type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly2.cs`. Le premier fichier contient une classe de base publique, `BaseClass`, et un type qui en est dérivé, `DerivedClass1`. `BaseClass` possède un membre protégé privé, `myValue`, auquel `DerivedClass1` tente d'accéder de deux manières. La première tentative

d'accès à `myValue` via une instance de `BaseClass` génère une erreur. Cependant, la tentative de l'utiliser comme un membre hérité dans `DerivedClass1` réussit.

Dans le deuxième fichier, une tentative d'accès à `myValue` en tant que membre hérité de `DerivedClass2` génère une erreur, car il est accessible seulement par des types dérivés dans Assembly1.

Si `Assembly1.cs` contient un InternalsVisibleToAttribute nom `Assembly2` , la classe dérivée aura `DerivedClass1` accès aux `private protected` membres déclarés dans `BaseClass` . `InternalsVisibleTo` rend `private protected` les membres visibles aux classes dérivées dans d'autres assemblys.

Les membres de struct ne peuvent pas être `private protected`, car le struct ne peut pas être hérité.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- intérieurs
- Problèmes de sécurité pour les mots clés virtuels internes

abstract (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Le modificateur `abstract` indique que l'élément en cours de modification a une implémentation manquante ou incomplète. Le modificateur abstract peut être utilisé avec des classes, des méthodes, des propriétés, des indexeurs et des événements. Dans une déclaration de classe, utilisez le modificateur `abstract` pour indiquer qu'une classe doit uniquement servir de classe de base pour d'autres classes, et ne pas être instanciée toute seule. Les membres définis comme abstraits doivent être implémentés par des classes non abstraites dérivées de la classe abstraite.

Exemple

Dans cet exemple, la classe `Square` doit fournir une implémentation de `GetArea`, car elle dérive de la classe `Shape` :

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Les classes abstraites présentent les caractéristiques suivantes :

- Une classe abstraite ne peut pas être instanciée.
- Une classe abstraite peut contenir des méthodes et accesseurs abstraits.
- Il n'est pas possible de modifier une classe abstraite avec le modificateur sealed, car les deux modificateurs ont des significations opposées. Le modificateur `sealed` empêche qu'une classe soit héritée et le modificateur `abstract` exige qu'une classe soit héritée.
- Une classe non abstraite dérivée d'une classe abstraite doit inclure des implémentations réelles de tous les accesseurs et méthodes abstraits hérités.

Dans une déclaration de méthode ou de propriété, utilisez le modificateur `abstract` pour indiquer que la méthode ou la propriété ne contient pas d'implémentation.

Les méthodes abstraites présentent les caractéristiques suivantes :

- Une méthode abstraite est implicitement une méthode virtuelle.

- Les déclarations de méthodes abstraites sont autorisées uniquement dans les classes abstraites.
- Comme une déclaration de méthode abstraite ne fournit pas d'implémentation réelle, il n'y a pas de corps de méthode ; la déclaration de méthode se termine simplement par un point-virgule, et la signature n'est pas suivie d'accolades ({ }). Par exemple :

  ```
  public abstract void MyMethod();
  ```

  L'implémentation est fournie par une méthode override, qui est membre d'une classe non abstraite.
- L'utilisation des modificateurs static ou virtual dans une déclaration de méthode abstraite serait une erreur.

Les propriétés abstraites se comportent comme les méthodes abstraites, à l'exception des différences dans la syntaxe de déclaration et d'appel.

- L'utilisation du modificateur `abstract` sur une propriété statique serait une erreur.
- Une propriété abstraite héritée peut être substituée dans une classe dérivée en incluant une déclaration de propriété qui utilise le modificateur override.

Pour plus d'informations sur les classes abstraites, consultez Classes abstract et sealed et membres de classe.

Une classe abstraite doit fournir une implémentation pour tous les membres d'interface.

Une classe abstraite qui implémente une interface peut mapper les méthodes d'interface à des méthodes abstraites. Par exemple :

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Exemple

Dans cet exemple, la classe `DerivedClass` est dérivée de la classe abstraite `BaseClass`. La classe abstraite contient une méthode abstraite, `AbstractMethod`, et deux propriétés abstraites, `X` et `Y`.

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();   // Abstract method
    public abstract int X    { get; }
    public abstract int Y    { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

Dans l'exemple précédent, si vous tentez d'instancier la classe abstraite en utilisant une instruction comme celle-ci :

```
BaseClass bc = new BaseClass();   // Error
```

Vous obtenez une erreur indiquant que le compilateur ne peut pas créer une instance de la classe abstraite BaseClass.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs

- virtual
- override
- Mots clés C#

async (référence C#)

02/11/2020 • 7 minutes to read • Edit Online

Utilisez le modificateur `async` pour spécifier qu'une méthode, une expression lambda ou une méthode anonyme sont asynchrones. Si vous utilisez ce modificateur sur une méthode ou une expression, il s'agit d'une *méthode async*. L'exemple suivant définit une méthode async nommée `ExampleMethodAsync` :

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

Si vous débutez en programmation asynchrone ou que vous ne comprenez pas comment une méthode Async utilise l' `await` opérateur pour effectuer un travail potentiellement long sans bloquer le thread de l'appelant, lisez l'introduction de la programmation asynchrone avec Async et await. Le code suivant se trouve dans une méthode async et appelle la méthode HttpClient.GetStringAsync :

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

Une méthode async s'exécute de façon synchrone jusqu'à ce qu'elle atteigne sa première expression `await`, où elle est suspendue jusqu'à ce que la tâche attendue soit terminée. Dans le même temps, le contrôle retourne à l'appelant de la méthode, comme le montre l'exemple indiqué dans la section suivante.

Si la méthode que le mot clé `async` modifie ne contient pas une expression ou une instruction `await`, la méthode s'exécute de façon synchrone. Un avertissement du compilateur vous signale toutes les méthodes async qui ne contiennent pas d'instructions `await`, car cette situation peut indiquer une erreur. Consultez Avertissement du compilateur (niveau 1) CS4014.

Le mot clé `async` est contextuel, car il est un mot clé uniquement lorsqu'il modifie une méthode, une expression lambda ou une méthode anonyme. Dans tous les autres contextes, il est interprété comme un identificateur.

Exemple

L'exemple suivant montre la structure et le flux de contrôle entre un gestionnaire d'événements asynchrones, `StartButton_Click`, et une méthode async, `ExampleMethodAsync`. Le résultat de la méthode async est le nombre de caractères d'une page web. Le code convient pour une application WPF (Windows Presentation Foundation) ou une application du Windows Store que vous créez dans Visual Studio ; consultez les commentaires du code pour configurer l'application.

Vous pouvez exécuter ce code dans Visual Studio en tant qu'application Windows Presentation Foundation (WPF) ou qu'application du Windows Store. Vous avez besoin d'un contrôle Button nommé `StartButton` et d'un contrôle Textbox nommé `ResultsTextBox`. N'oubliez pas de définir les noms et le gestionnaire afin d'obtenir un résultat semblable à ceci :

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top"
Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

Pour exécuter le code en tant qu'application WPF :

- Collez ce code dans la classe `MainWindow` dans MainWindow.xaml.cs.
- Ajoutez une référence à System.Net.Http.
- Ajoutez une directive `using` à System.Net.Http.

Pour exécuter le code comme une application du Windows Store :

- Collez ce code dans la classe `MainPage` dans MainPage.xaml.cs.
- Ajouter des directives using pour System.Net.Http et System.Threading.Tasks.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

Pour plus d'informations sur les tâches et le code qui s'exécute en attendant une tâche, consultez programmation asynchrone avec Async et await. Pour obtenir un exemple de console complet qui utilise des éléments semblables, consultez traiter des tâches asynchrones telles qu'elles sont terminées (C#).

Types de retour

Une méthode async peut avoir les types de retour suivants :

- Task
- Task<TResult>
- void. Les méthodes `async void` sont généralement déconseillées pour le code autre que les gestionnaires d'événements parce que les appelants ne peuvent pas `await` ces méthodes et doivent implémenter un mécanisme différent pour signaler les conditions d'erreur ou les complétions réussies.
- À compter de C# 7.0, tout type ayant une méthode `GetAwaiter` accessible. Le type `System.Threading.Tasks.ValueTask<TResult>` est une implémentation de ce genre. Il est disponible en ajoutant le package NuGet `System.Threading.Tasks.Extensions`.

La méthode async ne peut déclarer aucun paramètre in, ref ou out, ni avoir une valeur de retour de référence, mais elle peut appeler des méthodes qui ont ces paramètres.

Vous spécifiez `Task<TResult>` comme type de retour d'une méthode async si l'instruction return de la méthode spécifie un opérande de type `TResult`. Utilisez `Task` si aucune valeur significative n'est retournée lorsque la méthode est terminée. En d'autres termes, un appel à la méthode retourne `Task`, mais lorsque `Task` est terminé, toute expression `await` qui attend `Task` prend la valeur `void`.

Vous utilisez le type de retour `void` principalement pour définir les gestionnaires d'événements, qui ont besoin de ce type de retour. L'appelant d'une méthode async retournant `void` ne peut pas l'attendre et ne peut pas intercepter les exceptions levées par la méthode.

À partir de C# 7.0, vous retournez un autre type, en général un type valeur, qui a une méthode `GetAwaiter` permettant de limiter les allocations de mémoire dans les sections de code critiques pour les performances.

Pour obtenir plus d'informations et des exemples, consultez Types de retour Async.

Voir aussi

- AsyncStateMachineAttribute
- await
- Programmation asynchrone avec Async et await
- Traiter les tâches asynchrones terminées

const (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Vous utilisez le mot clé `const` pour déclarer un champ constant ou un élément local constant. Les champs et les éléments locaux constants ne sont pas des variables et ne peuvent pas être modifiés. Les constantes peuvent être des chiffres, des valeurs booléennes, des chaînes ou une référence null. Ne créez pas une constante pour représenter des informations qui doivent être modifiées. Par exemple, n'utilisez pas un champ constant pour stocker le prix d'un service, le numéro de version du produit ou le nom de la marque d'une société. Ces valeurs peuvent changer dans le temps, et dans la mesure où les compilateurs propagent les constantes, le code compilé avec vos bibliothèques devra être recompilé pour refléter ces modifications. Consultez également le mot clé readonly. Par exemple :

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Notes

Le type d'une déclaration constante indique le type des membres introduit par la déclaration. L'initialiseur d'un élément local constant ou d'un champ constant doit être une expression constante qui peut être implicitement convertie en type cible.

Une expression constante est une expression qui peut être complètement évaluée au moment de la compilation. C'est pourquoi, les seules valeurs possibles pour les constantes des types référence sont `string` et une référence null.

La déclaration constante peut déclarer plusieurs constantes, notamment :

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

Le modificateur `static` n'est pas autorisé dans une déclaration constante.

Une constante peut être utilisée dans une expression constante, comme suit :

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

Le mot clé readonly est différent du mot clé `const`. Un champ `const` ne peut être initialisé qu'au moment de la déclaration du champ. Un champ `readonly` peut être initialisé dans la déclaration ou dans un constructeur. C'est pourquoi, les champs `readonly` peuvent avoir des valeurs différentes en fonction du constructeur utilisé. De même, bien qu'un champ `const` soit une constante au moment de la compilation, le champ `readonly` peut être utilisé pour des constantes au moment de l'exécution, comme ci-après :

`public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Exemple

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

Exemple

Cet exemple montre comment utiliser des constantes en tant que variables locales.

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- seulement

Event (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Le mot clé `event` sert à déclarer un événement dans une classe d'éditeur.

Exemple

L'exemple suivant montre comment déclarer et déclencher un événement qui utilise EventHandler comme type délégué sous-jacent. Pour obtenir l'exemple de code complet qui montre également comment utiliser le EventHandler<TEventArgs> type délégué générique et comment s'abonner à un événement et créer une méthode de gestionnaire d'événements, consultez Comment publier des événements conformes aux instructions .net.

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Les événements constituent un type spécial de délégué multicast qui peut uniquement être appelé au sein de la classe ou du struct où ils sont déclarés (la classe d'éditeur). Si d'autres classes ou structs s'abonnent à l'événement, leurs méthodes de gestionnaire d'événements sont appelées quand la classe d'éditeur déclenche l'événement. Pour plus d'informations et pour obtenir des exemples de code, consultez Événements et Délégués.

Les événements peuvent être marqués comme public, Private, protected, Internal, protected internalou Private protected. Ces modificateurs d'accès définissent comment les utilisateurs de la classe peuvent accéder à l'événement. Pour plus d'informations, consultez Modificateurs d'accès.

Mots clés et événements

Les mots clés suivants s'appliquent aux événements.

MOT CLÉ	DESCRIPTION	INFORMATIONS SUPPLÉMENTAIRES
static	Rend le champ accessible à tout moment aux appelants, même s'il n'existe aucune instance de la classe.	Classes statiques et membres de classe statique

MOT CLÉ	DESCRIPTION	INFORMATIONS SUPPLÉMENTAIRES
virtual	Permet aux classes dérivées de substituer le comportement d'événement à l'aide du mot clé override.	Héritage
sealed	Spécifie que pour les classes dérivées l'événement n'est plus virtuel.	
abstraction	Le compilateur ne génère pas les blocs d'accesseurs d'événement `add` et `remove`, et par conséquent les classes dérivées doivent fournir leur propre implémentation.	

Un événement peut être déclaré comme événement statique à l'aide du mot clé static. Cela rend le champ accessible à tout moment aux appelants, même s'il n'existe aucune instance de la classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Un événement peut être marqué comme événement virtuel à l'aide du mot clé virtual. Cela permet aux classes dérivées de substituer le comportement d'événement à l'aide du mot clé override. Pour plus d'informations, consultez Héritage. Un événement qui se substitue à un événement virtuel peut également être sealed, ce qui signifie que pour les classes dérivées il n'est plus virtuel. Pour finir, un événement peut être déclaré abstract, ce qui signifie que le compilateur ne génère pas les blocs d'accesseurs d'événement `add` et `remove`. Ainsi, les classes dérivées doivent fournir leur propre implémentation.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- add
- remove
- Modificateurs
- Comment combiner des délégués (délégués multicast)

extern (référence C#)

16/12/2020 • 4 minutes to read • Edit Online

Le modificateur `extern` permet de déclarer une méthode qui est implémentée en externe. Le modificateur `extern` est souvent utilisé avec l'attribut `DllImport` lors de l'utilisation de services Interop à appeler dans du code non managé. Dans ce cas, la méthode doit également être déclarée comme `static`, comme indiqué dans l'exemple suivant :

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

Le mot clé `extern` peut également définir un alias d'assembly externe, permettant ainsi de référencer différentes versions du même composant à partir d'un seul assembly. Pour plus d'informations, consultez extern alias.

C'est une erreur d'utiliser conjointement les modificateurs abstract et `extern` pour modifier le même membre. L'utilisation du modificateur `extern` signifie que la méthode est implémentée en dehors du code C#, tandis que l'utilisation du modificateur `abstract` signifie que l'implémentation de la méthode n'est pas effectuée dans la classe.

L'utilisation du mot clé extern est plus restreinte dans C# que dans C++. Pour comparer son utilisation dans C# et C++, consultez : Utilisation d'extern pour spécifier la liaison dans le Guide de référence du langage C++.

Exemple 1

Dans cet exemple, le programme reçoit une chaîne provenant de l'utilisateur et l'affiche dans une boîte de message. Le programme utilise la méthode `MessageBox` importée de la bibliothèque User32.dll.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Exemple 2

Cet exemple illustre un programme C# qui fait appel à une bibliothèque C (une DLL native).

1. Créez le fichier C suivant et nommez-le `cmdll.c` :

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Ouvrez une fenêtre d'invite de commandes d'outils natifs Visual Studio x64 (ou x32) à partir du répertoire d'installation de Visual Studio et compilez le fichier `cmdll.c` en tapant **cl -LD cmdll.c** à l'invite de commandes.

3. Dans le même répertoire, créez le fichier C# suivant et nommez-le `cm.cs` :

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Ouvrez une fenêtre d'invite de commandes d'outils natifs Visual Studio x64 (ou x32) à partir du répertoire d'installation de Visual Studio et compilez le fichier `cm.cs` en tapant :

   > **csc cm.cs** (pour l'invite de commandes x64) —ou— **csc -platform:x86 cm.cs** (pour l'invite de commandes x32)

   Cette opération crée le fichier exécutable `cm.exe`.

5. Exécutez `cm.exe`. La méthode `SampleMethod` passe la valeur 5 au fichier DLL, qui retourne la valeur multipliée par 10. Le programme génère la sortie suivante :

```
SampleMethod() returns 50.
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- System.Runtime.InteropServices.DllImportAttribute
- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs

in (modificateur générique) (Référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Pour les paramètres de type générique, le mot clé `in` spécifie que le paramètre de type est contravariant. Vous pouvez utiliser le mot clé `in` dans les interfaces et délégués génériques.

La contravariance permet d'utiliser un type moins dérivé que celui spécifié par le paramètre générique. Cela permet la conversion implicite des classes qui implémentent des interfaces contravariantes, ainsi que la conversion implicite des types délégués. La covariance et la contravariance des paramètres de type générique sont prises en charge pour les types référence, mais pas pour les types valeur.

Un type peut être déclaré comme étant contravariant dans une interface ou un délégué générique uniquement s'il définit le type des paramètres d'une méthode et non le type de retour d'une méthode. Les paramètres `In`, `ref` et `out` doivent être invariants, ce qui signifie qu'ils sont ni covariants ni contravariants.

Une interface qui possède un paramètre de type contravariant permet à ses méthodes d'accepter des arguments de types moins dérivés que ceux spécifiés par le paramètre de type d'interface. Par exemple, dans l'interface IComparer<T>, le type T est contravariant, vous pouvez attribuer un objet du type `IComparer<Person>` à un objet du type `IComparer<Employee>` sans utiliser de méthode de conversion spéciale si `Employee` hérite `Person`.

Un délégué contravariant peut être assigné à un autre délégué du même type, mais avec un paramètre de type générique moins dérivé.

Pour plus d'informations, consultez Covariance et contravariance.

Interface générique contravariante

L'exemple suivant montre comment déclarer, étendre et implémenter une interface générique contravariante. Il montre également comment utiliser la conversion implicite pour les classes qui implémentent cette interface.

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Délégué générique contravariant

L'exemple de code suivant montre comment déclarer, instancier et invoquer un délégué générique contravariant. Il

montre également comment convertir implicitement un type délégué.

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- à
- Covariance et contravariance
- Modificateurs

new, modificateur (référence C#)

02/11/2020 • 6 minutes to read • Edit Online

En cas d'utilisation comme un modificateur de déclaration, le mot clé `new` masque explicitement un membre qui est hérité d'une classe de base. Lorsque vous masquez un membre hérité, la version dérivée du membre remplace la version de classe de base. Bien que vous puissiez masquer des membres sans utiliser le modificateur `new`, vous obtenez un avertissement du compilateur. Si vous utilisez `new` pour masquer explicitement un membre, il supprime cet avertissement.

Vous pouvez également utiliser le mot clé `new` pour créer une instance d'un type ou l'utiliser comme une contrainte de type générique.

Pour masquer un membre hérité, déclarez-le dans la classe dérivée en utilisant le même nom de membre, puis modifiez-le à l'aide du mot-clé `new`. Par exemple :

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

Dans cet exemple, `BaseC.Invoke` est masqué par `DerivedC.Invoke`. Le champ `x` n'est pas affecté parce qu'il n'est pas masqué par un nom semblable.

Le masquage de noms par le biais de l'héritage peut prendre l'une des formes suivantes :

- En général, une constante, un champ, une propriété ou un type qui est introduit dans une classe ou une structure masque tous les membres de la classe de base qui partagent son nom. Il existe des cas spéciaux. Par exemple, si vous déclarez un nouveau champ avec le nom `N` pour avoir un type qui n'est pas invocable, et qu'un type de base déclare `N` en tant que méthode, le nouveau champ ne masquera pas la déclaration de base dans la syntaxe d'appel. Pour plus d'informations, consultez la section Recherche de membres de la spécification du langage C#.
- Une méthode introduite dans une classe ou une structure masque les propriétés, les champs et les types qui partagent ce nom, dans la classe de base. Cela masque également toutes les méthodes de la classe de base ayant la même signature.
- Un indexeur introduit dans une classe ou un struct masque tous les indexeurs de la classe de base ayant la même signature.

L'utilisation des opérateurs `new` et override sur le même membre est une erreur, car ces deux modificateurs ont des significations qui s'excluent mutuellement. Le modificateur `new` crée un nouveau membre avec le même nom et entraîne le masquage du membre d'origine. Le modificateur `override` étend l'implémentation pour un membre hérité.

L'utilisation du modificateur `new` dans une déclaration qui ne masque pas un membre hérité génère un avertissement.

Exemple

Dans cet exemple, une classe de base, `BaseC` et une classe dérivée, `DerivedC`, utilisent le même nom de champ `x`, masquant ainsi la valeur du champ hérité. Cet exemple illustre l'utilisation du modificateur `new`. Il montre aussi comment accéder aux membres masqués de la classe de base en utilisant leurs noms complets.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

Exemple

Dans cet exemple, une classe imbriquée masque une classe du même nom dans la classe de base. Cet exemple illustre l'utilisation du modificateur `new` pour éliminer le message d'avertissement, ainsi que l'accès aux membres de la classe masquée à l'aide de leurs noms complets.

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

Si vous supprimez le modificateur `new`, le programme peut encore être compilé et exécuté, mais vous obtiendrez l'avertissement suivant :

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

spécification du langage C#

Pour plus d'informations, consultez la section Modificateur new de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- Versioning avec les mots clés override et new
- Savoir quand utiliser les mots clés override et new

out (modificateur générique) (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Pour les paramètres de type générique, le mot clé `out` spécifie que le paramètre de type est covariant. Vous pouvez utiliser le mot clé `out` dans les interfaces et délégués génériques.

La covariance permet d'utiliser un type plus dérivé que celui spécifié par le paramètre générique. Cela permet la conversion implicite des classes qui implémentent des interfaces covariantes, ainsi que la conversion implicite des types délégués. La covariance et la contravariance sont prises en charge pour les types référence, mais pas pour les types valeur.

Une interface qui possède un paramètre de type covariant permet à ses méthodes de retourner des types plus dérivés que ceux spécifiés par le paramètre de type. Par exemple, comme dans le .NET Framework 4, dans IEnumerable<T>, le type T est covariant, vous pouvez assigner un objet du type `IEnumerable(Of String)` à un objet du type `IEnumerable(Of Object)` sans utiliser de méthode de conversion spéciale.

Un délégué covariant peut être assigné à un autre délégué du même type, mais avec un paramètre de type générique plus dérivé.

Pour plus d'informations, consultez Covariance et contravariance.

Exemple - interface générique covariante

L'exemple suivant montre comment déclarer, étendre et implémenter une interface générique covariante. Il montre également comment utiliser la conversion implicite pour les classes qui implémentent une interface covariante.

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

Dans une interface générique, un paramètre de type peut être déclaré covariant s'il satisfait aux conditions suivantes :

- Le paramètre de type est utilisé uniquement comme type de retour des méthodes d'interface, mais pas comme type des arguments de méthode.

> **NOTE**
> Il existe une exception à cette règle. Si une interface covariante a un délégué générique contravariant comme paramètre de méthode, vous pouvez utiliser le type covariant comme paramètre de type générique pour ce délégué. Pour plus d'informations sur les délégués génériques covariants et contravariants, consultez Variance dans les délégués et Utilisation de la variance pour les délégués génériques Func et Action.

- Le paramètre de type n'est pas utilisé comme contrainte générique pour les méthodes d'interface.

Exemple - délégué générique covariant

L'exemple de code suivant montre comment déclarer, instancier et appeler un délégué générique covariant. Il montre également comment convertir implicitement des types délégués.

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

Dans un délégué générique, un type peut être déclaré comme étant covariant s'il est utilisé uniquement comme type de retour des méthodes, mais pas pour des arguments de méthode.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Variance dans les interfaces génériques
- in
- Modificateurs

override (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Le modificateur `override` est nécessaire pour étendre ou modifier l'implémentation abstraite ou virtuelle d'une méthode, d'une propriété, d'un indexeur ou d'un événement hérités.

Dans l'exemple suivant, la `Square` classe doit fournir une implémentation substituée de `GetArea` , car `GetArea` est héritée de la classe abstraite `Shape` :

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Une `override` méthode fournit une nouvelle implémentation de la méthode héritée d'une classe de base. La méthode substituée par une déclaration `override` est appelée méthode de base substituée. Une `override` méthode doit avoir la même signature que la méthode de base substituée. À compter de C# 9,0, les `override` méthodes prennent en charge les types de retour covariants. En particulier, le type de retour d'une `override` méthode peut dériver du type de retour de la méthode de base correspondante. Dans C# 8,0 et les versions antérieures, les types de retour d'une `override` méthode et la méthode de base substituée doivent être identiques.

Vous ne pouvez pas surcharger une méthode statique ou non virtuelle. La méthode de base substituée doit être `virtual` , `abstract` ou `override` .

Une déclaration `override` ne peut pas changer l'accessibilité de la méthode `virtual` . Les deux méthodes `override` et `virtual` doivent avoir le même modificateur de niveau d'accès.

Vous ne pouvez pas utiliser les modificateurs `new` , `static` ou `virtual` pour modifier une méthode `override` .

Une déclaration de propriété de substitution doit spécifier exactement le même modificateur d'accès, le même type et le même nom que la propriété héritée. À compter de C# 9,0, les propriétés de substitution en lecture seule prennent en charge les types de retour covariants. La propriété substituée doit être `virtual` , `abstract` ou `override` .

Pour plus d'informations sur l'utilisation du mot clé `override` , consultez Gestion de version avec les mots clés override et new et Savoir quand utiliser les mots clés override et new. Pour plus

d'informations sur l'héritage, consultez Héritage.

Exemple

Cet exemple définit une classe de base nommée `Employee` et une classe dérivée nommée `SalesEmployee`. La classe `SalesEmployee` inclut un champ supplémentaire (`salesbonus`) et substitue la méthode `CalculatePay` afin de la prendre en compte.

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                  decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice",
                      1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.name} earned: {employee2.CalculatePay()}");
    }
}
/*
    Output:
    Employee1 Alice earned: 1500
    Employee2 Bob earned: 1200
*/
```

spécification du langage C#

Pour plus d'informations, consultez la section méthodes de substitution de la spécification du langage C#.

Pour plus d'informations sur les types de retour covariants, consultez la Remarque relativeà la proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Héritage
- Mots clés C#
- Modificateurs
- abstraction
- virtuels
- new (modificateur)
- Polymorphisme

readonly (référence C#)

02/11/2020 • 7 minutes to read • Edit Online

Le `readonly` mot clé est un modificateur qui peut être utilisé dans quatre contextes :

- Dans une déclaration de champ, `readonly` indique que l'assignation au champ peut uniquement se produire dans le cadre de la déclaration ou dans un constructeur de la même classe. Un champ en lecture seule peut être affecté et réaffecté plusieurs fois dans la déclaration de champ et le constructeur.

  Un `readonly` champ ne peut pas être assigné après la sortie du constructeur. Cette règle a des implications différentes pour les types valeur et les types référence :

  - Étant donné que les types de valeur contiennent directement leurs données, un champ qui est un type de valeur `readonly` est immuable.
  - Étant donné que les types de référence contiennent une référence à leurs données, un champ qui est un type de référence `readonly` doit toujours faire référence au même objet. Cet objet n'est pas immuable. Le modificateur `readonly` empêche le champ d'être remplacé par une autre instance du type de référence. Toutefois, le modificateur n'empêche pas les données d'instance du champ d'être modifiées par le biais du champ en lecture seule.

  > **WARNING**
  > Un type visible de l'extérieur qui contient un champ en lecture seule visible de l'extérieur qui est un type référence mutable peut être une faille de sécurité et peut déclencher l'avertissement CA2104 : « ne déclarez pas les types référence mutables en lecture seule ».

- Dans une `readonly struct` définition de type, `readonly` indique que le type de structure est immuable. Pour plus d'informations, consultez la section `readonly` struct de l'article types de structures .

- Dans une déclaration de membre d'instance au sein d'un type structure, `readonly` indique qu'un membre d'instance ne modifie pas l'état de la structure. Pour plus d'informations, consultez la section `readonly` membres d'instance de l'article types de structures .

- Dans une `ref readonly` méthode retournée, le `readonly` modificateur indique que la méthode retourne une référence et que les écritures ne sont pas autorisées à cette référence.

Les `readonly struct` `ref readonly` contextes et ont été ajoutés en C# 7,2. `readonly` des membres de struct ont été ajoutés en C# 8,0

Exemple de champ en lecture seule

Dans cet exemple, la valeur du champ `year` ne peut pas être modifiée dans la méthode `ChangeYear` , même si une valeur lui est assignée dans le constructeur de classe :

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

Vous pouvez affecter une valeur à un champ `readonly` uniquement dans les contextes suivants :

- Lorsque la variable est initialisée dans la déclaration, par exemple :

  ```
  public readonly int y = 5;
  ```

- Dans un constructeur d'instance de la classe qui contient la déclaration de champ d'instance.
- Dans le constructeur statique de la classe qui contient la déclaration de champ statique.

Ces contextes de constructeur sont également les seuls contextes dans lesquels il est possible de passer un `readonly` champ en tant que paramètre out ou ref .

NOTE

Le mot clé `readonly` est différent du mot clé const. Un champ `const` ne peut être initialisé qu'au moment de la déclaration du champ. Un champ `readonly` peut être assigné plusieurs fois dans la déclaration de champ et dans un constructeur. C'est pourquoi, les champs `readonly` peuvent avoir des valeurs différentes en fonction du constructeur utilisé. De même, si un `const` champ est une constante de compilation, le `readonly` champ peut être utilisé pour les constantes Runtime, comme dans l'exemple suivant :

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

Dans l'exemple précédent, si vous utilisez une instruction telle que dans l'exemple suivant :

```
p2.y = 66;        // Error
```

vous obtiendrez le message d'erreur du compilateur :

Un champ ReadOnly ne peut pas être assigné (sauf s'il s'agit d'un constructeur ou d'un initialiseur de variable)

Exemple de retour ref readonly

Le `readonly` modificateur sur un `ref return` indique que la référence retournée ne peut pas être modifiée. L'exemple suivant retourne une référence à l'origine. Elle utilise le `readonly` modificateur pour indiquer que les appelants ne peuvent pas modifier l'origine :

```
private static readonly SamplePoint origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref origin;
```

Le type retourné ne doit pas nécessairement être un `readonly struct`. Tout type pouvant être retourné par `ref` peut être retourné par `ref readonly`.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de

référence pour la syntaxe C# et son utilisation.

Vous pouvez également consulter les propositions de spécification de langage :

- ReadOnly et struct ReadOnly
- membres de struct ReadOnly

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- const
- Fields

sealed (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Lorsqu'il est appliqué à une classe, le modificateur `sealed` empêche les autres classes d'en hériter. Dans l'exemple suivant, la classe `B` hérite de la classe `A`, mais aucune classe ne peut hériter de la classe `B`.

```
class A {}
sealed class B : A {}
```

Vous pouvez également utiliser le modificateur `sealed` sur une méthode ou une propriété qui substitue une méthode ou une propriété virtuelle dans une classe de base. Ainsi, vous pouvez autoriser les classes à dériver de votre classe et les empêcher de substituer des méthodes ou des propriétés virtuelles spécifiques.

Exemple

Dans l'exemple suivant, `Z` hérite de `Y` mais `Z` ne peut pas substituer la fonction virtuelle `F` qui est déclarée dans `X` et scellée (sealed) dans `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Lorsque vous définissez de nouvelles méthodes ou propriétés dans une classe, vous pouvez empêcher les classes dérivées de les substituer en ne les déclarant pas comme virtuelles.

Une erreur consiste à utiliser le modificateur abstract avec une classe sealed, car une classe abstraite doit être héritée par une classe qui fournit une implémentation des méthodes ou des propriétés abstraites.

Lorsqu'il est appliqué à une méthode ou une propriété, le modificateur `sealed` doit toujours être utilisé avec override.

Comme les structs sont implicitement sealed, ils ne peuvent pas être hérités.

Pour plus d'informations, consultez Héritage.

Pour plus d'exemples, consultez Classes abstract et sealed et membres de classe.

Exemple

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

Dans l'exemple précédent, vous pouvez essayer d'hériter de la classe sealed à l'aide de l'instruction suivante :

```
class MyDerivedC: SealedClass {} // Error
```

Le résultat est un message d'erreur :

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Remarques

Pour déterminer s'il faut sceller une classe, une méthode ou une propriété, vous devez généralement prendre en compte les deux points suivants :

- Les avantages potentiels dont les classes dérivées peuvent bénéficier grâce à la possibilité de personnaliser votre classe.
- Le risque que les classes dérivées puissent modifier vos classes de telle manière qu'elles ne fonctionnent plus correctement ou comme prévu.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Classes statiques et membres de classe statique
- Classes abstract et sealed et membres de classe
- Modificateurs d'accès
- Modificateurs
- remplacer
- virtual

static (référence C#)

02/11/2020 • 6 minutes to read • Edit Online

Cette page couvre le `static` mot clé de modificateur. Le `static` mot clé fait également partie de la `using static` directive.

Utilisez le modificateur `static` pour déclarer un membre statique, qui appartient au type lui-même plutôt qu'à un objet spécifique. Le `static` modificateur peut être utilisé pour déclarer des `static` classes. Dans les classes, les interfaces et les structs, vous pouvez ajouter le `static` modificateur à des champs, des méthodes, des propriétés, des opérateurs, des événements et des constructeurs. Le `static` modificateur ne peut pas être utilisé avec des indexeurs ou des finaliseurs. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

À compter de C# 8,0, vous pouvez ajouter le `static` modificateur à une fonction locale. Une fonction locale statique ne peut pas capturer les variables locales ou l'état de l'instance.

À compter de C# 9,0, vous pouvez ajouter le `static` modificateur à une expression lambda ou une méthode anonyme. Une méthode lambda statique ou anonyme ne peut pas capturer les variables locales ou l'état de l'instance.

Exemple : classe statique

La classe suivante est déclarée comme `static` et contient uniquement des méthodes `static` :

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

Une déclaration de constante ou de type est implicitement un `static` membre. Un `static` membre ne peut pas être référencé par le biais d'une instance. Au lieu de cela, elle est référencée par le nom de type. Par exemple, considérons la classe suivante :

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

Pour faire référence au `static` membre `x` , utilisez le nom qualifié complet, `MyBaseC.MyStruct.x` , sauf si le membre est accessible à partir de la même portée :

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

Alors qu'une instance d'une classe contient une copie distincte de tous les champs d'instance de la classe, il n'existe qu'une seule copie de chaque `static` champ.

Il n'est pas possible d'utiliser `this` pour référencer des `static` méthodes ou des accesseurs de

propriété.

Si le `static` mot clé est appliqué à une classe, tous les membres de la classe doivent avoir la valeur `static` .

Les classes, les interfaces et les `static` classes peuvent avoir des `static` constructeurs. Un `static` constructeur est appelé à un moment donné entre le démarrage du programme et l'instanciation de la classe.

> **NOTE**
> L'utilisation du mot clé `static` est plus restreinte que dans C++. Pour comparer avec le mot clé C++, consultez Classes de stockage (C++).

Pour illustrer `static` les membres, considérez une classe qui représente un employé de la société. Supposons que la classe contient une méthode pour compter les employés et un champ pour stocker le nombre d'employés. La méthode et le champ n'appartiennent pas à une instance d'employé. Au lieu de cela, ils appartiennent à la classe des employés dans son ensemble. Ils doivent être déclarés en tant que `static` membres de la classe.

Exemple : champ et méthode statiques

Cet exemple lit le nom et l'ID d'un nouvel employé, incrémente d'une unité le compteur d'employés et affiche les informations concernant le nouvel employé et le nouveau nombre d'employés. Ce programme lit le nombre actuel d'employés à partir du clavier.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Exemple : initialisation statique

Cet exemple montre que vous pouvez initialiser un `static` champ à l'aide d'un autre `static` champ qui n'est pas encore déclaré. Les résultats ne sont pas définis tant que vous n'affectez pas explicitement une valeur au `static` champ.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- using static, directive
- Classes statiques et membres de classe statique

unsafe (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `unsafe` désigne un contexte non sécurisé, qui est requis pour toute opération impliquant des pointeurs. Pour plus d'informations, consultez Pointeurs et code unsafe.

Vous pouvez utiliser le modificateur `unsafe` dans la déclaration d'un type ou d'un membre. Toute l'étendue de texte du type ou du membre est ainsi considérée comme un contexte unsafe. Par exemple, ce qui suit est une méthode déclarée avec le modificateur `unsafe` :

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

La portée du contexte unsafe s'étend de la liste de paramètres à la fin de la méthode, de sorte que les pointeurs peuvent également être utilisés dans la liste de paramètres :

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

Vous pouvez également avoir recours à un bloc unsafe pour utiliser un code unsafe dans ce bloc. Par exemple :

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Pour compiler du code unsafe, vous devez spécifier l' `-unsafe` option du compilateur. Le code unsafe n'est pas vérifiable par le service CLR (Common Language Runtime).

Exemple

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

spécification du langage C#

Pour plus d'informations, voir Code unsafe dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- fixed, instruction
- Pointeurs et code unsafe
- Mémoires tampons de taille fixe

virtual (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Le mot clé `virtual` sert à modifier une méthode, une propriété, un indexeur ou une déclaration event et leur permet d'être substitués dans une classe dérivée. Par exemple, cette méthode peut être substituée par toute classe qui en hérite :

```
public virtual double Area()
{
    return x * y;
}
```

L'implémentation d'un membre virtuel peut être modifiée par un membre de substitution dans une classe dérivée. Pour plus d'informations sur l'utilisation du `virtual` mot clé, consultez contrôle de version avec les mots clés override et New et savoir quand utiliser les mots clés override et New.

Remarques

Quand une méthode virtuelle est appelée, un membre de substitution est recherché dans le type d'objet au moment de l'exécution. Le membre de substitution de la classe la plus dérivée est appelé (cela peut être le membre d'origine), si aucune classe dérivée n'a substitué le membre.

Par défaut, les méthodes ne sont pas virtuelles. Vous ne pouvez pas substituer une méthode non virtuelle.

Vous ne pouvez pas utiliser le modificateur `virtual` avec les modificateurs `static`, `abstract`, `private` ou `override`. L'exemple suivant illustre une propriété virtuelle :

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

Les propriétés virtuelles se comportent comme des méthodes virtuelles, à l'exception des différences dans la syntaxe de déclaration et d'appel.

- L'utilisation du modificateur `virtual` sur une propriété statique serait une erreur.
- Une propriété virtuelle héritée peut être substituée dans une classe dérivée en incluant une déclaration de propriété qui utilise le modificateur `override`.

Exemple

Dans cet exemple, la classe `Shape` contient les deux coordonnées `x` et `y`, ainsi que la méthode virtuelle `Area()`. Différentes classes de formes, telles que `Circle`, `Cylinder` et `Sphere`, héritent de la classe `Shape`, et la surface est calculée pour chaque figure. Chaque classe dérivée a sa propre implémentation de substitution de `Area()`.

Notez que les classes héritées `Circle`, `Sphere` et `Cylinder` utilisent toutes des constructeurs qui initialisent la classe de base, comme indiqué dans la déclaration suivante.

```
public Cylinder(double r, double h): base(r, h) {}
```

Le programme suivant calcule et affiche la zone appropriée pour chaque figure en appelant l'implémentation appropriée de la méthode `Area()`, selon l'objet associé à la méthode.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
```

```
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Polymorphisme
- abstraction
- remplacer
- new (modificateur)

volatile (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Le mot clé `volatile` indique qu'un champ peut être modifié par plusieurs threads qui s'exécutent simultanément. Le compilateur, le système de runtime et même le matériel sont susceptibles de réorganiser les lectures et les écritures sur des emplacements de mémoire pour des raisons de performances. Les champs déclarés `volatile` ne sont pas soumis à ces optimisations. Le fait d'ajouter le modificateur `volatile` permet de garantir que tous les threads observent les écritures volatiles effectuées par un autre thread dans l'ordre dans lequel elles ont été effectuées. Rien ne garantit que les écritures volatiles présentent un ordre total unique, tel que le voient tous les threads d'exécution.

Le mot clé `volatile` peut être appliqué aux champs des types suivants :

- Types référence.
- Types pointeur (dans un contexte unsafe). Notez que, même si le pointeur lui-même peut être volatile, l'objet sur lequel il pointe ne le peut pas. En d'autres termes, vous ne pouvez pas déclarer un pointeur vers un objet volatile.
- Types simples comme `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` et `bool`.
- Type `enum` avec l'un des types de base suivants : `byte`, `sbyte`, `short`, `ushort`, `int` ou `uint`.
- Paramètres de type générique connus comme des types référence.
- Voir IntPtr et UIntPtr.

Les autres types, notamment `double` et `long`, ne peuvent pas être marqués `volatile`, car il n'y a aucune garantie que les lectures et écritures sur des champs de ce type soient atomiques. Pour protéger l'accès multithread à ces types de champs, utilisez les membres de la Interlocked classe ou Protégez l'accès à l'aide de l' `lock` instruction.

Le mot clé `volatile` ne peut s'appliquer qu'aux champs d'une `class` ou d'un `struct`. Les variables locales ne peuvent pas être déclarées `volatile`.

Exemple

L'exemple ci-dessous montre comment déclarer une variable de champ public comme `volatile`.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int _i)
    {
        sharedStorage = _i;
    }
}
```

L'exemple suivant montre comment il est possible de créer un thread auxiliaire ou de travail et de l'utiliser pour effectuer le traitement en parallèle avec le thread principal. Pour plus d'informations sur le multithreading, voir Threading managé.

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

Si vous ajoutez le modificateur `volatile` à la déclaration de `_shouldStop` en place, vous obtiendrez toujours les mêmes résultats (similaires à l'extrait indiqué dans le code précédent). Sans ce modificateur sur le membre `_shouldStop` en revanche, le comportement est imprévisible. La méthode `DoWork` peut optimiser l'accès au membre, ce qui entraîne la lecture de données périmées. En raison de la nature de la programmation multithread, le nombre de lectures obsolètes est imprévisible. Différentes exécutions du programme produiront des résultats légèrement différents.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Spécification du langage C# : mot clé volatile
- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- Lock (instruction)
- Interlocked

Mots clés d'instruction (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Les instructions sont des instructions de programme. Sauf mention contraire dans les rubriques référencées dans le tableau suivant, les instructions sont exécutées en séquence. Le tableau suivant répertorie les mots clés d'instructions C#. Pour plus d'informations sur les instructions exprimées sans mots clés, consultez Instructions.

CATEGORY	MOTS CLÉS C#
Instructions de sélection	if, else, switch, case
Instructions d'itération	do, for, foreach, in, while
Instructions de saut	break, continue, default, goto, return, yield
Instructions de gestion des exceptions	throw, try-catch, try-finally, try-catch-finally
Activé et désactivé	checked, unchecked
Fixed, instruction	des
Lock (instruction)	Lock

Voir aussi

- Référence C#
- Instructions
- Mots clés C#

if-else (référence C#)

02/11/2020 • 8 minutes to read • Edit Online

Une instruction `if` identifie l'instruction à exécuter en fonction de la valeur d'une expression booléenne. Dans l'exemple suivant, la variable `bool``condition` est définie sur `true` puis archivé dans l'instruction `if` . La sortie est `The variable is set to true.` .

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

Vous pouvez exécuter les exemples de cette rubrique en les plaçant dans la méthode `Main` d'une application console.

En C#, une instruction `if` peut prendre deux formes, comme le montre l'exemple suivant.

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

Dans une instruction `if-else` , si `condition` a la valeur true, `then-statement` s'exécute. Si `condition` a la valeur false, `else-statement` s'exécute. Sachant que `condition` ne peut pas avoir simultanément les valeurs true et false, `then-statement` et `else-statement` d'une instruction `if-else` ne peuvent jamais s'exécuter. Une fois que `then-statement` ou `else-statement` s'est exécuté, le contrôle est transféré à l'instruction suivante après l'instruction `if` .

Dans une instruction `if` qui n'inclut pas d'instruction `else` , si `condition` a la valeur true, `then-statement` s'exécute. Si `condition` a la valeur false, le contrôle est transféré à l'instruction suivante après l'instruction `if` .

`then-statement` et `else-statement` peuvent tous deux être constitués d'une ou plusieurs instructions placées entre accolades (`{}`). Pour une seule instruction, les accolades sont facultatives, mais recommandées.

La ou les instructions contenues dans `then-statement` et `else-statement` peuvent être de n'importe quel type, y compris une autre instruction `if` imbriquée à l'intérieur de l'instruction `if` d'origine. Dans les instructions

`if` imbriquées, chaque clause `else` appartient à la dernière instruction `if` qui n'a pas d'instruction `else` correspondante. Dans l'exemple suivant, `Result1` s'affiche si `m > 10` et `n > 20` ont tous deux la valeur true. Si `m > 10` a la valeur true, mais que `n > 20` a la valeur false, `Result2` s'affiche.

```
// Try with m = 12 and then with m = 8.
int m = 12;
int n = 18;

if (m > 10)
    if (n > 20)
    {
        Console.WriteLine("Result1");
    }
    else
    {
        Console.WriteLine("Result2");
    }
```

Si vous préférez que `Result2` s'affiche quand `(m > 10)` a la valeur false, vous pouvez spécifier cette association au moyen d'accolades pour établir le début et la fin de l'instruction `if` imbriquée, comme le montre l'exemple suivant.

```
// Try with m = 12 and then with m = 8.
if (m > 10)
{
    if (n > 20)
        Console.WriteLine("Result1");
}
else
{
    Console.WriteLine("Result2");
}
```

`Result2` s'affiche si la condition `(m > 10)` a la valeur false.

Exemple

Dans l'exemple suivant, vous entrez un caractère au clavier et le programme utilise une instruction `if` imbriquée pour déterminer si le caractère d'entrée est un caractère alphabétique. Si le caractère d'entrée est un caractère alphabétique, le programme vérifie si c'est une minuscule ou une majuscule. Un message s'affiche dans chaque cas.

```
Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.IsLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.
```

Exemple

Vous pouvez également imbriquer une `if` instruction à l'intérieur d'un bloc Else, comme le montre le code partiel suivant. L'exemple imbrique des instructions `if` à l'intérieur de deux blocs « else » et d'un bloc « then ». Les commentaires précisent les conditions qui sont vraies (true) et celles qui sont fausses (false) dans chaque bloc.

```
// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}
```

Exemple

L'exemple suivant détermine si un caractère d'entrée est une lettre minuscule, une lettre majuscule ou un nombre. Si ces trois conditions ont la valeur false, le caractère n'est pas un caractère alphanumérique. L'exemple affiche un message dans chaque cas.

```
Console.Write("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

De la même manière que le bloc « else » ou le bloc « then » peuvent contenir n'importe quelle instruction valide, vous pouvez utiliser n'importe quelle expression booléenne valide pour la condition. Vous pouvez utiliser des opérateurs logiques tels que `!` ,,,, `&&` `||` `&` `|` et `^` pour créer des conditions composées. Le code suivant présente des exemples.

```
// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- ?:, Opérateur
- if-else, instruction (C++)
- switch

switch (informations de référence sur C#)

02/11/2020 • 25 minutes to read • Edit Online

Cet article traite de l' `switch` instruction. Pour plus d'informations sur l' `switch` expression (introduite dans C# 8,0), consultez l'article sur les `switch` expressions dans la section expressions et opérateurs .

`switch` est une instruction de sélection qui choisit une *section à commutateur* unique à exécuter à partir d'une liste de candidats en fonction d'une correspondance de modèle avec l' *expression de correspondance*.

```
using System;

public class Example
{
   public static void Main()
   {
      int caseSwitch = 1;

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
              Console.WriteLine("Case 2");
              break;
          default:
              Console.WriteLine("Default case");
              break;
      }
   }
}
// The example displays the following output:
//       Case 1
```

L'instruction `switch` est souvent utilisée comme alternative à une construction if-else si une expression unique est testée en fonction de trois conditions ou plus. Par exemple, l'instruction `switch` suivante détermine laquelle des trois valeurs possibles a été affectée à une variable de type `Color` :

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      switch (c)
      {
         case Color.Red:
            Console.WriteLine("The color is red");
            break;
         case Color.Green:
            Console.WriteLine("The color is green");
            break;
         case Color.Blue:
            Console.WriteLine("The color is blue");
            break;
         default:
            Console.WriteLine("The color is unknown.");
            break;
      }
   }
}
```

Elle est équivalente à l'exemple suivant, qui utilise une construction `if` - `else`.

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      if (c == Color.Red)
         Console.WriteLine("The color is red");
      else if (c == Color.Green)
         Console.WriteLine("The color is green");
      else if (c == Color.Blue)
         Console.WriteLine("The color is blue");
      else
         Console.WriteLine("The color is unknown.");
   }
}
// The example displays the following output:
//       The color is red
```

Expression de correspondance

L'expression de correspondance fournit la valeur à mettre en correspondance avec les modèles dans les étiquettes `case`. Sa syntaxe est la suivante :

```
   switch (expr)
```

Avec C# 6 (et les versions antérieures), l'expression de correspondance doit retourner une valeur d'un des types suivants :

- char,
- chaîne.
- bool,
- valeur intégrale , telle qu'un `int` ou un `long` .
- valeur enum .

À compter de C# 7.0, l'expression de correspondance peut être toute expression non Null.

Section de commutation

Une instruction `switch` inclut une ou plusieurs sections de commutation. Chaque section Switch contient un ou plusieurs *noms de cas* (un cas ou une étiquette par défaut) suivis d'une ou plusieurs instructions. L'instruction `switch` peut inclure au maximum une étiquette par défaut, placée dans n'importe quelle section de commutation. L'exemple suivant montre une instruction `switch` simple qui a trois sections de commutation, chacune contenant à son tour deux instructions. La deuxième section de commutation contient les étiquettes `case 2:` et `case 3:` .

Une instruction `switch` peut inclure un nombre quelconque de sections de commutation, et chaque section peut contenir une ou plusieurs étiquettes case, comme dans l'exemple ci-dessous. Toutefois, deux étiquettes case ne doivent pas contenir la même expression.

```
using System;

public class Example
{
   public static void Main()
   {
      Random rnd = new Random();
      int caseSwitch = rnd.Next(1,4);

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
          case 3:
              Console.WriteLine($"Case {caseSwitch}");
              break;
          default:
              Console.WriteLine($"An unexpected value ({caseSwitch})");
              break;
      }
   }
}
// The example displays output like the following:
//       Case 1
```

Une seule section de commutation s'exécute dans une instruction switch. C# ne permet pas à l'exécution de passer d'une section switch à la suivante. Pour cette raison, le code suivant génère une erreur du compilateur, CS0163 : « Le contrôle ne peut pas passer d'une étiquette case (<case label>) à une autre ».

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

Si cela est nécessaire, il est possible de procéder en quittant explicitement la section de commutation à l'aide d'une instruction break, goto ou return. Toutefois, le code suivant est également valide, car il garantit que le contrôle du programme ne peut pas passer à la section switch `default`.

```
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
           Console.WriteLine("Endless looping. . . .");
    default:
        Console.WriteLine("Default value...");
        break;
}
```

L'exécution de la liste d'instructions dans la section de commutation avec une étiquette case qui correspond à l'expression de correspondance commence avec la première instruction et continue en suivant la liste d'instructions, en général jusqu'à ce qu'une instruction de saut, telle que `break`, `goto case`, `goto label`, `return` ou `throw`, soit atteinte. À ce stade, le contrôle est transféré hors de l'instruction `switch` ou vers un autre nom de cas. Si une instruction `goto` est utilisée, elle doit transférer le contrôle à une étiquette constante. Cette restriction est nécessaire, car tenter de transférer le contrôle à une étiquette non constante peut avoir des effets indésirables, tels que le transfert du contrôle à un emplacement inattendu dans le code ou la création d'une boucle sans fin.

Étiquettes case

Chaque étiquette case spécifie un modèle à comparer à l'expression de correspondance (la variable `caseSwitch` dans les exemples précédents). S'ils correspondent, le contrôle est transféré à la section de commutation qui contient la **première** étiquette case correspondante. Si aucun modèle d'étiquette case ne correspond à l'expression de correspondance, le contrôle est transféré à la section comportant l'étiquette case `default`, s'il y en a une. En l'absence de cas `default`, aucune instruction d'aucune section switch n'est exécutée et le contrôle est transféré hors de l'instruction `switch`.

Pour plus d'informations sur l'instruction `switch` et les critères spéciaux, consultez la section Critères spéciaux avec l'instruction `switch`.

Étant donné que C# 6 ne prend en charge que le modèle de constante et n'autorise pas la répétition de valeurs constantes, les étiquettes case définissent des valeurs qui s'excluent mutuellement, et un seul modèle peut correspondre à l'expression utilisée. Par conséquent, l'ordre dans lequel les instructions `case` apparaissent n'a pas d'importance.

Dans C# 7.0, toutefois, comme d'autres modèles sont pris en charge, les étiquettes case ne sont pas tenues de définir des valeurs s'excluant mutuellement et plusieurs modèles peuvent correspondre à l'expression de correspondance. Comme seules les instructions de la première section de commutation contenant le modèle correspondant sont exécutées, l'ordre dans lequel les instructions `case` apparaissent est désormais important. Si C# détecte une section de commutation dont la ou les instructions case sont équivalentes aux instructions précédentes, ou en sont des sous-ensembles, C# génère une erreur du compilateur, CS8120, « Le switch case a déjà été pris en charge par un case antérieur ».

L'exemple suivant illustre une instruction `switch` qui utilise divers modèles ne s'excluant pas mutuellement. Si l'on déplace la section switch `case 0:` de sorte qu'elle n'est plus la première section de l'instruction `switch`, C# génère une erreur du compilateur, car un entier dont la valeur est égale à zéro est un sous-ensemble de tous les entiers, ce qui correspond au modèle défini par l'instruction `case int val`.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Example
{
   public static void Main()
   {
      var values = new List<object>();
      for (int ctr = 0; ctr <= 7; ctr++) {
         if (ctr == 2)
            values.Add(DiceLibrary.Roll2());
         else if (ctr == 4)
            values.Add(DiceLibrary.Pass());
         else
            values.Add(DiceLibrary.Roll());
      }

      Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
   }
}

public static class DiceLibrary
{
   // Random number generator to simulate dice rolls.
   static Random rnd = new Random();

   // Roll a single die.
   public static int Roll()
   {
      return rnd.Next(1, 7);
   }

   // Roll two dice.
   public static List<object> Roll2()
   {
      var rolls = new List<object>();
      rolls.Add(Roll());
      rolls.Add(Roll());
      return rolls;
   }

   // Calculate the sum of n dice rolls.
   public static int DiceSum(IEnumerable<object> values)
   {
      var sum = 0;
      foreach (var item in values)
      {
            switch (item)
            {
```

```
                // A single zero value.
                case 0:
                   break;
                // A single value.
                case int val:
                   sum += val;
                   break;
                // A non-empty collection.
                case IEnumerable<object> subList when subList.Any():
                   sum += DiceSum(subList);
                   break;
                // An empty collection.
                case IEnumerable<object> subList:
                   break;
                //  A null reference.
                case null:
                   break;
                // A value that is neither an integer nor a collection.
                default:
                   throw new InvalidOperationException("unknown item type");
             }
       }
       return sum;
    }

    public static object Pass()
    {
       if (rnd.Next(0, 2) == 0)
          return null;
       else
          return new List<object>();
    }
 }
```

Vous pouvez corriger ce problème et éliminer l'avertissement du compilateur de deux façons :

- en modifiant l'ordre des sections de commutation ;
- en utilisant une clause when dans l'étiquette `case`.

Étiquette case `default`

L'étiquette case `default` spécifie la section switch à exécuter si l'expression utilisée ne correspond à aucune autre étiquette `case`. En l'absence de cas `default`, si l'expression ne correspond à aucune autre étiquette `case`, le flux de programme passe à travers l'instruction `switch`.

L'étiquette case `default` peut apparaître à n'importe quelle position dans l'instruction `switch`. Quelle que soit sa position dans le code source, il est toujours évalué en dernier, une fois que toutes les étiquettes `case` ont été évaluées.

Critères spéciaux avec l'instruction `switch`

Chaque instruction `case` définit un modèle qui, s'il correspond à l'expression de correspondance, entraîne l'exécution de la section de commutation qui le contient. Toutes les versions de C# prennent en charge le modèle de constante. Les autres modèles sont pris en charge à compter de C# 7.0.

Modèle de constante

Le modèle de constante teste si l'expression de correspondance est égale à une constante spécifiée. Sa syntaxe est la suivante :

```
    case constant:
```

où *constant* est la valeur à tester. *constant* peut correspondre à l'une des expressions constantes suivantes :

- Littéral bool : `true` ou `false` .
- Toute constante intégrale , telle qu'un `int` , un `long` ou un `byte` .
- Le nom d'une variable `const` déclarée
- Une constante d'énumération
- Un littéral de type char
- Un littéral de type string

L'expression constante est évaluée de la manière suivante :

- Si *expr* et *constant* sont des types intégraux, l'opérateur d'égalité C# détermine si l'expression retourne `true` (autrement dit, si `expr == constant`).
- Sinon, la valeur de l'expression est déterminée par un appel à la méthode statique Object.Equals(expr, constant).

L'exemple suivant utilise le modèle de constante pour déterminer si une date particulière correspond à un jour de week-end, au premier jour de la semaine, au dernier jour de la semaine de travail ou au milieu de la semaine de travail. Il évalue la propriété DateTime.DayOfWeek du jour actuel par rapport aux membres de l'énumération DayOfWeek.

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
           case DayOfWeek.Sunday:
           case DayOfWeek.Saturday:
              Console.WriteLine("The weekend");
              break;
           case DayOfWeek.Monday:
              Console.WriteLine("The first day of the work week.");
              break;
           case DayOfWeek.Friday:
              Console.WriteLine("The last day of the work week.");
              break;
           default:
              Console.WriteLine("The middle of the work week.");
              break;
        }
    }
}
// The example displays output like the following:
//       The middle of the work week.
```

L'exemple suivant utilise le modèle de constante pour gérer l'entrée d'utilisateur dans une application console qui simule une machine à café automatique.

```
using System;

class Example
{
   static void Main()
   {
       Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
       Console.Write("Please enter your selection: ");
       string str = Console.ReadLine();
       int cost = 0;

       // Because of the goto statements in cases 2 and 3, the base cost of 25
       // cents is added to the additional cost for the medium and large sizes.
       switch (str)
       {
          case "1":
          case "small":
              cost += 25;
              break;
          case "2":
          case "medium":
              cost += 25;
              goto case "1";
          case "3":
          case "large":
              cost += 50;
              goto case "1";
          default:
              Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
              break;
      }
      if (cost != 0)
      {
          Console.WriteLine("Please insert {0} cents.", cost);
      }
      Console.WriteLine("Thank you for your business.");
   }
}
// The example displays output like the following:
//         Coffee sizes: 1=small 2=medium 3=large
//         Please enter your selection: 2
//         Please insert 50 cents.
//         Thank you for your business.
```

Modèle de type

Le modèle de type permet une évaluation et une conversion rapides de type. Lorsqu'il est utilisé avec l'instruction `switch` pour effectuer une mise en correspondance de modèle, il permet de tester si une expression peut être convertie en un type spécifié et, si tel est le cas, il effectue un cast de l'expression en une variable de ce type. Sa syntaxe est la suivante :

```
   case type varname
```

où *type* est le nom du type vers lequel le résultat de *expr* doit être converti, et *varname* est l'objet vers lequel le résultat de *expr* est converti si la correspondance est établie. À compter de C# 7.1, le type *expr* au moment de la compilation peut être un paramètre de type générique.

L'expression `case` est `true` si l'une quelconque des affirmations suivantes est vraie :

- *expr* est une instance du même type que *type*.
- *expr* est une instance d'un type qui dérive de *type*. En d'autres termes, le résultat de *expr* peut être upcasté en une instance de *type*.

- *expr* a un type au moment de la compilation qui est une classe de base de *type* et *expr* a un type au moment de l'exécution égal à *type* ou dérivé de *type*. Le *type au moment de la compilation* d'une variable est le type de la variable, tel qu'il est défini dans sa déclaration de type. Le *type au moment de l'exécution* d'une variable est le type de l'instance qui est assignée à cette variable.
- *expr* est une instance d'un type qui implémente l'interface *type*.

Si l'expression case est true, *varname* est définitivement assigné et a une portée locale au sein de la section de commutation uniquement.

Notez que `null` ne correspond pas à un type. Pour mettre en correspondance `null`, vous utilisez l'étiquette `case` suivante :

```
case null:
```

L'exemple suivant utilise le modèle de type pour fournir des informations sur différentes sortes de types de collection.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
               Console.WriteLine($"An array with {arr.Length} elements.");
               break;
            case IEnumerable<int> ieInt:
               Console.WriteLine($"Average: {ieInt.Average(s => s)}");
               break;
            case IList list:
               Console.WriteLine($"{list.Count} items");
               break;
            case IEnumerable ie:
               string result = "";
               foreach (var e in ie)
                  result += $"{e} ";
               Console.WriteLine(result);
               break;
            case null:
               // Do nothing for a null.
               break;
            default:
               Console.WriteLine($"A instance of type {coll.GetType().Name}");
               break;
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

Au lieu de `object`, on pourrait créer une méthode générique en utilisant le type de la collection comme paramètre de type, comme le montre le code suivant :

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation<T>(T coll)
    {
        switch (coll)
        {
            case Array arr:
               Console.WriteLine($"An array with {arr.Length} elements.");
               break;
            case IEnumerable<int> ieInt:
               Console.WriteLine($"Average: {ieInt.Average(s => s)}");
               break;
            case IList list:
               Console.WriteLine($"{list.Count} items");
               break;
            case IEnumerable ie:
               string result = "";
               foreach (var e in ie)
                  result += $"{e} ";
               Console.WriteLine(result);
               break;
            case object o:
               Console.WriteLine($"A instance of type {o.GetType().Name}");
               break;
            default:
                Console.WriteLine("Null passed to this method.");
                break;
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
//     Null passed to this method.
```

La version générique présente deux différences avec le premier exemple. Tout d'abord, on ne peut pas utiliser le cas `null`. Aucun cas constant n'est possible, car le compilateur ne peut pas convertir un type arbitraire `T` en un type autre que `object`. Ce qui était auparavant le cas `default` teste maintenant la présence d'un `object` non Null, ce qui signifie que le cas `default` ne concerne que les valeurs `null`.

Sans critères spéciaux, ce code peut être écrit comme suit. L'utilisation de critères spéciaux de type génère un code plus compact et lisible en éliminant la nécessité de tester si le résultat d'une conversion est un `null` et d'effectuer des casts répétés.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array)
        {
           Array arr = (Array) coll;
           Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>)
        {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList)
        {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable)
        {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var e in ie)
               result += $"{e} ";
            Console.WriteLine(result);
        }
        else if (coll == null)
        {
            // Do nothing.
        }
        else
        {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

Instruction `case` et clause `when`

À compter de C# 7.0, comme les instructions case ne s'excluent pas nécessairement mutuellement, vous pouvez ajouter une clause `when` pour spécifier une condition supplémentaire qui doit être satisfaite pour que l'instruction case soit évaluée à true. La clause `when` peut être toute expression qui retourne une

valeur booléenne.

L'exemple suivant définit une classe `Shape` de base, une classe `Rectangle` qui dérive de `Shape` et une classe `Square` qui dérive de `Rectangle`. Il utilise la clause `when` pour que `ShowShapeInfo` traite un objet `Rectangle` dont la longueur et la largeur sont égales à celles d'un `Square`, même s'il n'a pas été instancié comme objet `Square`. La méthode ne tente pas d'afficher des informations sur un objet `null` ou sur une forme dont l'aire est nulle.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public override double Circumference
   {
      get { return 2 * Math.PI * Radius; }
   }

   public override double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
```

```
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                         sh, new Square(0), new Rectangle(8, 8),
                         new Circle(3) };
      foreach (var shape in shapes)
         ShowShapeInfo(shape);
   }

   private static void ShowShapeInfo(Shape sh)
   {
      switch (sh)
      {
         // Note that this code never evaluates to true.
         case Shape shape when shape == null:
            Console.WriteLine($"An uninitialized shape (shape == null)");
            break;
         case null:
            Console.WriteLine($"An uninitialized shape");
            break;
         case Shape shape when sh.Area == 0:
            Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
            break;
         case Square sq when sh.Area > 0:
            Console.WriteLine("Information about square:");
            Console.WriteLine($"   Length of a side: {sq.Side}");
            Console.WriteLine($"   Area: {sq.Area}");
            break;
         case Rectangle r when r.Length == r.Width && r.Area > 0:
            Console.WriteLine("Information about square rectangle:");
            Console.WriteLine($"   Length of a side: {r.Length}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Rectangle r when sh.Area > 0:
            Console.WriteLine("Information about rectangle:");
            Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Shape shape when sh != null:
            Console.WriteLine($"A {sh.GetType().Name} shape");
            break;
         default:
            Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
            break;
      }
   }
}
// The example displays the following output:
//       Information about square:
//          Length of a side: 10
//          Area: 100
//       Information about rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       An uninitialized shape
//       The shape: Square with no dimensions
//       Information about square rectangle:
//          Length of a side: 8
//          Area: 64
//       A Circle shape
```

Notez que la clause `when` de l'exemple qui tente de tester si un objet `Shape` est `null` ne s'exécute pas. Le

modèle de type correct à utiliser pour tester un `null` est `case null:`.

spécification du langage C#

Pour plus d'informations, consultez la section Instruction switch dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- if-else
- Critères spéciaux

do (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction `do` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`. Dans la mesure où cette expression est évaluée après chaque exécution de la boucle, une boucle `do-while` s'exécute une ou plusieurs fois. Cela diffère de la boucle while, qui s'exécute zéro ou plusieurs fois.

À tout moment dans le bloc d'instructions `do`, vous pouvez sortir de la boucle à l'aide de l'instruction break.

Vous pouvez passer directement à l'évaluation de l'expression `while` à l'aide de l'instruction continue. Si l'expression correspond à `true`, l'exécution passe à la première instruction de la boucle. Sinon, l'exécution passe à la première instruction après la boucle.

Vous pouvez également quitter une `do-while` boucle à l'aide des instructions goto, Returnou throw .

Exemple

L'exemple suivant illustre l'utilisation de l'instruction `do`. Sélectionnez **Exécuter** pour exécuter l'exemple de code. Après cela, vous pouvez modifier le code et le réexécuter.

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

spécification du langage C#

Pour plus d'informations, voir la section Instruction do de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- while, instruction

for (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

L'instruction `for` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`.

À tout moment dans le bloc d'instructions `for`, vous pouvez sortir de la boucle à l'aide de l'instruction break, ou passer à l'itération suivante de la boucle à l'aide de l'instruction continue. Vous pouvez également quitter une `for` boucle à l'aide des instructions goto, Returnou throw .

Structure de l'instruction `for`

Chaque instruction `for` définit des sections *initialiseur*, *condition* et *itérateur* :

```
for (initializer; condition; iterator)
    body
```

Ces trois sections sont facultatives. Le corps de la boucle est une instruction ou un bloc d'instructions.

L'exemple suivant montre l'instruction `for` avec toutes les sections définies :

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

La section *initialiseur*

Les instructions figurant dans la section *initialiseur* sont exécutées une seule fois, avant d'entrer dans la boucle. La section *initialiseur* contient au choix :

- La déclaration et l'initialisation d'une variable de boucle locale, qui n'est pas accessible à partir de l'extérieur de la boucle, ou
- Zéro, une ou plusieurs expressions d'instruction parmi celles de la liste suivante, séparées par des virgules :
  - Instruction d'assignation
  - Appel d'une méthode
  - Expression d'incrémentation préfixée ou suffixée, telle que `++i` ou `i++`
  - Expression de décrémentation préfixée ou suffixée, telle que `--i` ou `i--`
  - Création d'un objet à l'aide de l'opérateur new
  - Expression await

La section *initialiseur* dans l'exemple ci-dessus déclare et initialise la variable de boucle locale `i` :

```
int i = 0
```

La section *condition*

La section *condition*, si elle est présente, doit être une expression booléenne. Cette expression est évaluée avant chaque itération de boucle. Si la section *condition* est absente ou que l'expression booléenne est `true`, l'itération suivante de la boucle est exécutée ; sinon, la boucle est fermée.

La section *condition* dans l'exemple ci-dessus détermine si la boucle se termine en fonction de la valeur de la variable de boucle locale :

```
i < 5
```

La section *itérateur*

La section *itérateur* définit ce qui se produit après chaque itération du corps de la boucle. La section *itérateur* contient zéro, une ou plusieurs des expressions d'instruction suivantes, séparées par des virgules :

- Instruction d'assignation
- Appel d'une méthode
- Expression d'incrémentation préfixée ou suffixée, telle que `++i` ou `i++`
- Expression de décrémentation préfixée ou suffixée, telle que `--i` ou `i--`
- Création d'un objet à l'aide de l'opérateur new
- Expression await

La section *itérateur* dans l'exemple ci-dessus incrémente la variable de boucle locale :

```
i++
```

Exemples

L'exemple suivant illustre plusieurs utilisations moins courantes des sections d'instruction `for` : l'attribution d'une valeur à une variable de boucle externe dans la section *initialiseur*, l'appel d'une méthode dans les sections *initialiseur* et *itérateur*, et la modification des valeurs de deux variables dans la section *itérateur*. Sélectionnez **Exécuter** pour exécuter l'exemple de code. Après cela, vous pouvez modifier le code et le réexécuter.

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i=
{i}, j={j}"))
{
    // Body of the loop.
}
```

L'exemple suivant définit la boucle `for` infinie :

```
for ( ; ; )
{
    // Body of the loop.
}
```

spécification du langage C#

Pour plus d'informations, voir la section Instruction for de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- foreach, in

foreach, instruction (C#)

02/11/2020 • 6 minutes to read • Edit Online

L' `foreach` instruction exécute une instruction ou un bloc d'instructions pour chaque élément d'une instance du type qui implémente l' System.Collections.IEnumerable System.Collections.Generic.IEnumerable<T> interface ou, comme le montre l'exemple suivant :

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    count++;
    Console.WriteLine($"Element #{count}: {element}");
}
Console.WriteLine($"Number of elements: {count}");
```

L' `foreach` instruction n'est pas limitée à ces types. Vous pouvez l'utiliser avec une instance de n'importe quel type répondant aux conditions suivantes :

- Un type a la méthode sans paramètre public `GetEnumerator` dont le type de retour est de type classe, struct ou interface. À compter de C# 9,0, la `GetEnumerator` méthode peut être la méthode d'extensiond'un type.
- Le type de retour de la `GetEnumerator` méthode a la `Current` propriété publique et la méthode sans paramètre public `MoveNext` dont le type de retour est Boolean .

L'exemple suivant utilise l' `foreach` instruction avec une instance du System.Span<T> type, qui n'implémente aucune interface :

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

À compter de C# 7,3, si la propriété de l'énumérateur `Current` retourne une valeur de retour de référence (`ref T` où `T` est le type d'un élément de collection), vous pouvez déclarer une variable d'itération avec le `ref` `ref readonly` modificateur ou, comme le montre l'exemple suivant :

```
public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.Write($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}
```

À compter de C# 8,0, vous pouvez utiliser l' `await foreach` instruction pour consommer un flux de données asynchrone, autrement dit, le type de collection qui implémente l' IAsyncEnumerable<T> interface. Chaque itération de la boucle peut être interrompue pendant que l'élément suivant est récupéré de manière asynchrone. L'exemple suivant montre comment utiliser l' `await foreach` instruction :

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Par défaut, les éléments de flux sont traités dans le contexte capturé. Si vous souhaitez désactiver la capture du contexte, utilisez la TaskAsyncEnumerableExtensions.ConfigureAwait méthode d'extension. Pour plus d'informations sur les contextes de synchronisation et la capture du contexte actuel, consultez utilisation du modèle asynchrone basé sur des tâches. Pour plus d'informations sur les flux asynchrones, consultez la section Streams asynchrones de l'article nouveautés de C# 8,0 .

À tout moment dans le bloc d'instructions `foreach`, vous pouvez sortir de la boucle à l'aide de l'instruction break, ou passer à l'itération suivante de la boucle à l'aide de l'instruction continue. Vous pouvez également quitter une `foreach` boucle à l'aide des instructions goto, Returnou throw .

Si l'instruction `foreach` est appliquée à `null`, une NullReferenceException est levée. Si la collection source de l' `foreach` instruction est vide, le corps de la `foreach` boucle n'est pas exécuté et ignoré.

Type d'une variable d'itération

Vous pouvez utiliser le `var` mot clé pour permettre au compilateur de déduire le type d'une variable d'itération dans l' `foreach` instruction, comme le montre le code suivant :

```
foreach (var item in collection) { }
```

Vous pouvez également spécifier explicitement le type d'une variable d'itération, comme le montre le code suivant :

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

Dans le formulaire précédent, le type `T` d'un élément de collection doit être implicitement ou explicitement convertible en type `V` d'une variable d'itération. Si une conversion explicite de `T` en `V` échoue au moment de l'exécution, l' `foreach` instruction lève une exception InvalidCastException . Par exemple, si `T` est un type de classe non scellé, `V` peut être n'importe quel type d'interface, même celui qui `T` n'est pas implémenté. Au moment de l'exécution, le type d'un élément de collection peut être celui qui dérive de `T` et implémente réellement `V` . Si ce n'est pas le cas, une InvalidCastException exception est levée.

spécification du langage C#

Pour plus d'informations, voir la section Instruction foreach de la spécification du langage C#.

Pour plus d'informations sur les fonctionnalités ajoutées dans C# 8,0 et versions ultérieures, consultez les notes de proposition de fonctionnalités suivantes :

- Flux asynchrones (C# 8,0)
- `GetEnumerator` Prise en charge des extensions pour les `foreach` boucles (C# 9,0)

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Utilisation de foreach avec des tableaux
- for, instruction

while (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction `while` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`. Dans la mesure où cette expression est évaluée avant chaque exécution de la boucle, une boucle `while` s'exécute plusieurs fois ou pas du tout. Cela diffère de la boucle do, qui s'exécute une ou plusieurs fois.

À tout moment dans le bloc d'instructions `while`, vous pouvez sortir de la boucle à l'aide de l'instruction break.

Vous pouvez passer directement à l'évaluation de l'expression `while` à l'aide de l'instruction continue. Si l'expression correspond à `true`, l'exécution passe à la première instruction de la boucle. Sinon, l'exécution passe à la première instruction après la boucle.

Vous pouvez également quitter une `while` boucle à l'aide des instructions goto, Returnou throw .

Exemple

L'exemple suivant illustre l'utilisation de l'instruction `while`. Sélectionnez **Exécuter** pour exécuter l'exemple de code. Après cela, vous pouvez modifier le code et le réexécuter.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

spécification du langage C#

Pour plus d'informations, voir la section Instruction while de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- instruction do

break (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

L'instruction `break` termine la boucle englobante ou l'instruction switch la plus proche dans laquelle elle figure. Le contrôle est passé à l'instruction qui suit l'instruction terminée, le cas échéant.

Exemple

Dans cet exemple, l'instruction conditionnelle contient un compteur qui est supposé compter de 1 à 100. Toutefois, l'instruction `break` termine la boucle après le chiffre 4.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
 Output:
    1
    2
    3
    4
*/
```

Exemple

Cet exemple illustre l'utilisation de `break` dans une instruction switch.

```
class Switch
{
    static void Main()
    {
        Console.Write("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is 1");
                break;
            case 2:
                Console.WriteLine("Current value is 2");
                break;
            case 3:
                Console.WriteLine("Current value is 3");
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/
```

Si vous aviez entré `4`, le résultat serait le suivant :

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

Exemple

Dans cet exemple, l'instruction `break` est utilisée pour sortir d'une boucle imbriquée interne et passer le contrôle à la boucle externe. Le contrôle n'est retourné *qu'* un niveau dans les boucles imbriquées.

```
class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop.
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"num = {numbers[i]}");

            // Inner loop.
            for (int j = 0; j < letters.Length; j++)
            {
                if (j == i)
                {
                    // Return control to outer loop.
                    break;
                }
                Console.Write($" {letters[j]} ");
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    num = 0

    num = 1
     a
    num = 2
     a  b
    num = 3
     a  b  c
    num = 4
     a  b  c  d
    num = 5
     a  b  c  d  e
    num = 6
     a  b  c  d  e  f
    num = 7
     a  b  c  d  e  f  g
    num = 8
     a  b  c  d  e  f  g  h
    num = 9
     a  b  c  d  e  f  g  h  i
 */
```

Exemple

Dans cet exemple, l' `break` instruction est utilisée uniquement pour sortir de la branche active au cours de chaque itération de la boucle. La boucle elle-même n'est pas affectée par les instances de `break` qui appartiennent à l'instruction switch imbriquée.

```
class BreakFromSwitchInsideLoop
{
    static void Main(string[] args)
    {
        // loop 1 to 3
        for (int i = 1; i <= 3; i++)
        {
            switch(i)
            {
                case 1:
                    Console.WriteLine("Current value is 1");
                    break;
                case 2:
                    Console.WriteLine("Current value is 2");
                    break;
                case 3:
                    Console.WriteLine("Current value is 3");
                    break;
                default:
                    Console.WriteLine("This shouldn't happen.");
                    break;
            }
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    Current value is 1
    Current value is 2
    Current value is 3
 */
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- switch

continue (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction `continue` passe le contrôle à l'itération suivante de l'instruction while, do, for ou foreach englobante dans laquelle elle apparaît.

Exemple

Dans cet exemple, un compteur est initialisé pour compter de 1 à 10. En utilisant l' `continue` instruction conjointement avec l'expression `(i < 9)` , les instructions entre `continue` et la fin du `for` corps sont ignorées dans les itérations où `i` est inférieur à 9. Au cours des deux dernières itérations de la `for` boucle (où i = = 9 et i = = 10), l' `continue` instruction n'est pas exécutée et la valeur de `i` est imprimée sur la console.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
9
10
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Break (instruction)

goto (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction `goto` transfère le contrôle du programme directement à une instruction étiquetée.

Une utilisation courante de `goto` consiste à transférer le contrôle à une étiquette switch-case ou à l'étiquette par défaut d'une instruction `switch`.

L'instruction `goto` sert aussi à quitter des boucles fortement imbriquées.

Exemple

L'exemple suivant montre comment utiliser `goto` dans une instruction switch.

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine($"Please insert {cost} cents.");
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input:  2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

Exemple

L'exemple suivant montre comment utiliser `goto` pour quitter des boucles imbriquées.

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array.
        for (int i = 0; i < x; i++)

            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input.
        Console.Write("Enter the number to search for: ");

        // Input a string.
        string myNumber = Console.ReadLine();

        // Search.
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine($"The number {myNumber} was not found.");
        goto Finish;

    Found:
        Console.WriteLine($"The number {myNumber} is found.");

    Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- goto, instruction (C++)

return (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'instruction `return` met un terme à l'exécution de la méthode dans laquelle elle apparaît et retourne le contrôle à la méthode d'appel. Elle peut également retourner une valeur facultative. Si la méthode est un type `void`, l'instruction `return` peut être omise.

Si l'instruction return est à l'intérieur d'un bloc `try`, le bloc `finally`, le cas échéant, est exécuté avant que le contrôle retourne à la méthode d'appel.

Exemple

Dans l'exemple suivant, la méthode `CalculateArea()` retourne la variable locale `area` en tant que valeur `double`.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Instruction return

throw (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Signale l'occurrence d'une exception pendant l'exécution du programme.

Remarques

La syntaxe de `throw` est :

```
throw [e];
```

où `e` est une instance d'une classe dérivée de System.Exception. L'exemple suivant utilise l'instruction `throw` pour lever une exception IndexOutOfRangeException si l'argument passé à une méthode nommée `GetNumber` ne correspond pas à un index valide d'un tableau interne.

```
using System;

public class NumberGenerator
{
   int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

   public int GetNumber(int index)
   {
      if (index < 0 || index >= numbers.Length) {
         throw new IndexOutOfRangeException();
      }
      return numbers[index];
   }
}
```

Les appelants de méthode utilisent alors un bloc `try-catch` ou `try-catch-finally` pour gérer l'exception levée. L'exemple suivant gère l'exception levée par la méthode `GetNumber`.

```
using System;

public class Example
{
   public static void Main()
   {
      var gen = new NumberGenerator();
      int index = 10;
      try {
          int value = gen.GetNumber(index);
          Console.WriteLine($"Retrieved {value}");
      }
      catch (IndexOutOfRangeException e)
      {
         Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
      }
   }
}
// The example displays the following output:
//        IndexOutOfRangeException: 10 is outside the bounds of the array
```

Exception levée plusieurs fois

`throw` peut également être utilisé dans un bloc `catch` pour lever de nouveau une exception gérée dans un bloc `catch`. Dans ce cas, `throw` n'accepte pas d'opérande d'exception. Cela est très utile lorsqu'une méthode passe un argument d'un appelant à une autre méthode de bibliothèque, et que la méthode de bibliothèque lève une exception qui doit être passée à l'appelant. Par exemple, l'exemple suivant lève de nouveau une exception NullReferenceException qui est levée lorsque vous tentez de récupérer le premier caractère d'une chaîne non initialisée.

```
using System;

public class Sentence
{
   public Sentence(string s)
   {
      Value = s;
   }

   public string Value { get; set; }

   public char GetFirstCharacter()
   {
      try {
         return Value[0];
        }
      catch (NullReferenceException e) {
         throw;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var s = new Sentence(null);
      Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
   }
}
// The example displays the following output:
//    Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//       at Sentence.GetFirstCharacter()
//       at Example.Main()
```

IMPORTANT

Vous pouvez également utiliser la syntaxe `throw e` dans un bloc `catch` pour instancier une nouvelle exception que vous passez à l'appelant. Dans ce cas, la trace de la pile de l'exception d'origine, qui est disponible à partir de la propriété StackTrace, n'est pas conservée.

Expression `throw`

À compter de C# 7.0, `throw` peut être utilisé comme expression et comme instruction. Ainsi, une exception peut être levée dans des contextes qui n'étaient précédemment pas pris en charge. notamment :

- opérateur conditionnel. l'exemple suivant utilise une expression `throw` pour lever une exception ArgumentException si une méthode reçoit un tableau de chaînes vide. Avant C# 7.0, cette logique devait figurer dans une instruction `if` / `else`.

```
private static void DisplayFirstNumber(string[] args)
{
   string arg = args.Length >= 1 ? args[0] :
                              throw new ArgumentException("You must supply an argument");
   if (Int64.TryParse(arg, out var number))
      Console.WriteLine($"You entered {number:F0}");
   else
      Console.WriteLine($"{arg} is not a number.");
}
```

- l'opérateur de fusion de Null : dans l'exemple suivant, une expression `throw` est utilisée avec un opérateur de fusion de Null pour lever une exception si la chaîne assignée à une propriété `Name` est `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- un lambda ou une méthode expression-bodied : l'exemple suivant illustre une méthode expression-bodied qui lève une exception InvalidCastException, car une conversion vers une valeur DateTime n'est pas prise en charge.

```
DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- try-catch
- Mots clés C#
- Comment : lever explicitement des exceptions

try-catch (référence C#)

02/11/2020 • 14 minutes to read • Edit Online

L'instruction try-catch consiste en un bloc `try` suivi d'une ou plusieurs clauses `catch` qui spécifient des gestionnaires pour différentes exceptions.

Quand une exception est levée, le Common Language Runtime (CLR) recherche l'instruction `catch` qui gère cette exception. Si la méthode en cours d'exécution ne contient pas un tel bloc `catch`, le CLR examine la méthode qui a appelé la méthode actuelle, puis remonte la pile des appels. Si aucun bloc `catch` n'est trouvé, alors le CLR affiche un message d'exception non gérée à l'utilisateur et arrête l'exécution du programme.

Le bloc `try` contient le code protégé susceptible de provoquer l'exception. Le bloc est exécuté jusqu'à ce qu'une exception soit levée ou qu'il se soit correctement terminé. Par exemple, la tentative suivante d'effectuer un cast d'un objet `null` déclenche l'exception NullReferenceException :

```
object o2 = null;
try
{
    int i2 = (int)o2;   // Error
}
```

Bien que la clause `catch` puisse être utilisée sans arguments pour intercepter tout type d'exception, cette utilisation est déconseillée. En général, vous devez intercepter uniquement les exceptions desquelles vous savez comment récupérer. Par conséquent, vous devez toujours spécifier un argument d'objet dérivé de System.Exception, par exemple :

```
catch (InvalidCastException e)
{
}
```

Il est possible d'utiliser plusieurs clauses `catch` spécifiques dans la même instruction try-catch. Dans ce cas, l'ordre des clauses `catch` est important car les clauses `catch` sont examinées dans l'ordre. Interceptez les exceptions plus spécifiques avant les moins spécifiques. Le compilateur produit une erreur si vous organisez vos blocs catch de sorte qu'un bloc ultérieur ne puisse jamais être atteint.

L'utilisation d'arguments `catch` constitue un moyen de filtrer les exceptions à gérer. Vous pouvez également utiliser un filtre d'exception qui examine l'exception pour déterminer si elle doit être prise en charge. Si le filtre d'exception retourne la valeur false, la recherche d'un gestionnaire se poursuit.

```
catch (ArgumentException e) when (e.ParamName == "…")
{
}
```

L'utilisation de filtres d'exceptions est préférable à une interception et une nouvelle levée (voir explication ci-dessous), car les filtres laissent la pile intact. Si un gestionnaire ultérieur vide la pile, vous pouvez déterminer d'où l'exception provient à l'origine, au lieu de déterminer simplement le dernier emplacement auquel elle a été levée. Une utilisation courante des expressions de filtre d'exception est liée à la journalisation. Vous pouvez créer un filtre qui retourne toujours false et dont la sortie est journalisée. Vous pouvez journaliser les exceptions au fur et à mesure sans avoir à les prendre en charge et à les lever de

nouveau.

Une instruction throw peut être utilisée dans un bloc `catch` pour lever une nouvelle fois l'exception interceptée par l'instruction `catch`. L'exemple suivant extrait des informations sources d'une exception IOException, puis lève l'exception à la méthode parente.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

Vous pouvez intercepter une seule exception et lever une exception différente. Dans ce cas, spécifiez l'exception que vous interceptez en tant qu'exception interne, comme illustré dans l'exemple suivant.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

Vous pouvez également lever à nouveau une exception quand une condition spécifiée a la valeur true, comme illustré dans l'exemple suivant.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

Il est également possible d'utiliser un filtre d'exception pour obtenir un résultat similaire d'une manière souvent plus claire (sans modifier la pile, comme expliqué précédemment dans ce document). L'exemple suivant a un comportement similaire à l'exemple précédent pour les appelants. La fonction lève et retourne `InvalidCastException` à l'appelant quand `e.Data` a une valeur `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

Dans un bloc `try`, initialisez uniquement les variables qui y sont déclarées. Sinon, une exception peut se produire avant la fin de l'exécution du bloc. Par exemple, dans l'exemple de code suivant, la variable `n` est

initialisée à l'intérieur du bloc `try`. Une tentative d'utilisation de cette variable en dehors du bloc `try` dans l'instruction `Write(n)` génère une erreur du compilateur.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

Pour plus d'informations sur l'interception, consultez try-catch-finally.

Exceptions dans les méthodes async

Une méthode async est marquée par un modificateur async et contient généralement une ou plusieurs expressions ou instructions await. Une expression await applique l'opérateur await à un Task ou Task<TResult>.

Quand le contrôle atteint un `await` dans la méthode async, la progression de la méthode est interrompue jusqu'à ce que la tâche attendue se termine. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode. Pour plus d'informations, consultez l'article Programmation asynchrone avec async et await.

La tâche terminée à laquelle `await` est appliqué peut être dans un état d'erreur en raison d'une exception non gérée dans la méthode qui retourne la tâche. L'attente de la tâche lève une exception. Une tâche peut également se terminer dans un état annulé si le processus asynchrone qui la retourne est annulé. L'attente d'une tâche annulée lève une exception `OperationCanceledException` .

Pour intercepter l'exception, attendez la tâche dans un bloc `try`, puis interceptez l'exception dans le bloc `catch` associé. Pour obtenir un exemple, consultez la section Exemple de la méthode async.

Une tâche peut être dans un état d'erreur car plusieurs exceptions se sont produites dans la méthode async attendue. Par exemple, la tâche peut être le résultat d'un appel à Task.WhenAll. Quand vous attendez une telle tâche, une seule des exceptions est interceptée et vous ne pouvez pas prévoir laquelle. Pour obtenir un exemple, consultez la section Exemple Task.WhenAll.

Exemple

Dans l'exemple suivant, le bloc `try` contient un appel à la méthode `ProcessString` qui risque de provoquer une exception. La clause `catch` clause contient le gestionnaire d'exceptions qui affiche simplement un message à l'écran. Quand l'instruction `throw` est appelée depuis `ProcessString`, le système recherche l'instruction `catch` et affiche le message `Exception caught`.

```
class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
   at TryFinallyTest.Main() Exception caught.
 * */
```

Exemple de deux blocs catch

Dans l'exemple suivant, deux blocs catch sont utilisés, et l'exception la plus spécifique, qui apparaît la première, est interceptée.

Pour intercepter l'exception la moins spécifique, vous pouvez remplacer l'instruction throw dans `ProcessString` par l'instruction suivante : `throw new Exception()`.

Si vous placez le bloc catch le moins spécifique en premier dans l'exemple, le message d'erreur suivant s'affiche : `A previous catch clause already catches all exceptions of this or a super type ('System.Exception')`.

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
 Output:
 System.ArgumentNullException: Value cannot be null.
 at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/
```

Exemple de la méthode async

L'exemple suivant illustre la gestion des exceptions pour les méthodes async. Pour intercepter une exception levée par une tâche async, placez l'expression `await` dans un bloc `try` et interceptez-la dans un bloc `catch`.

Supprimez les marques de commentaire de la ligne `throw new Exception` dans l'exemple pour illustrer la gestion des exceptions. La propriété `IsFaulted` de la tâche a la valeur `True`, la propriété `Exception.InnerException` de la tâche a la valeur de l'exception et l'exception est interceptée dans le bloc `catch`.

Supprimez les marques de commentaire de la ligne `throw new OperationCanceledException` pour montrer ce qui se passe quand vous annulez un processus asynchrone. La propriété `IsCanceled` de la tâche a la valeur `true` et l'exception est interceptée dans le bloc `catch`. Sous certaines conditions qui s'appliquent à cet exemple, la propriété `IsFaulted` de la tâche a la valeur `true` et `IsCanceled` a la valeur `false`.

```
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

Exemple Task.WhenAll

L'exemple suivant illustre la gestion des exceptions quand plusieurs tâches peuvent entraîner plusieurs exceptions. Le bloc `try` attend la tâche retournée par un appel à Task.WhenAll. La tâche est terminée quand les trois tâches auxquelles WhenAll est appliqué sont terminées.

Chacune de ces trois tâches provoque une exception. Le bloc `catch` itère au sein des exceptions, qui sont trouvent dans la propriété `Exception.InnerExceptions` de la tâche retournée par Task.WhenAll.

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

spécification du langage C#

Pour plus d'informations, consultez la section Instruction try de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Instructions try, throw et catch (C++)
- throw
- try-finally
- Comment : lever explicitement des exceptions

try-finally (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

En utilisant un bloc `finally`, vous pouvez nettoyer toutes les ressources allouées dans un bloc try et vous pouvez exécuter du code même si une exception se produit dans le bloc `try`. En règle générale, les instructions d'un bloc `finally` s'exécutent lorsque le contrôle quitte une instruction `try`. Le transfert de contrôle peut se produire suite à une exécution normale, à l'exécution d'une instruction `break`, `continue`, `goto` ou `return`, ou à la propagation d'une exception hors de l'instruction `try`.

Dans une exception gérée, le bloc `finally` associé est assuré d'être exécuté. Toutefois, si l'exception n'est pas gérée, l'exécution du bloc `finally` dépend de la manière dont l'opération de déroulement d'exception est déclenchée. Ceci, à son tour, dépend du paramétrage de votre ordinateur.

En général, lorsqu'une exception non gérée met fin à une application, que le bloc `finally` soit exécuté ou non n'est pas important. Toutefois, si vous avez des instructions dans un bloc `finally` qui doivent être exécutées même dans cette situation, une solution consiste à ajouter un bloc `catch` à l'instruction `try` - `finally`. Ou bien, vous pouvez intercepter l'exception qui peut être levée dans le bloc `try` d'une instruction `try` - `finally` plus haut dans la pile des appels. Autrement dit, vous pouvez intercepter l'exception dans la méthode qui appelle la méthode contenant l'instruction `try` - `finally`, ou dans la méthode qui appelle cette méthode, ou dans n'importe quelle méthode figurant dans la pile des appels. Si l'exception n'est pas interceptée, l'exécution du bloc `finally` varie selon que le système d'exploitation choisit de déclencher une opération de déroulement d'exception.

Exemple

Dans l'exemple suivant, une instruction de conversion non valide provoque une exception `System.InvalidCastException`. L'exception n'est pas gérée.

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

Dans l'exemple suivant, une exception issue de la méthode `TryCast` est interceptée dans une méthode plus loin dans la pile des appels.

```
public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    public static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

Pour plus d'informations sur `finally`, consultez try-catch-finally.

C# contient également l'instruction using, qui fournit des fonctionnalités similaires pour les objets IDisposable dans une syntaxe pratique.

spécification du langage C#

Pour plus d'informations, consultez la section Instruction try de la spécification du langage C#.

Voir aussi

- Référence C#

- Guide de programmation C#
- Mots clés C#
- Instructions try, throw et catch (C++)
- throw
- try-catch
- Comment : lever explicitement des exceptions

try-catch-finally (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`catch` et `finally` sont souvent utilisés ensemble pour obtenir et utiliser des ressources dans un bloc `try`, pour traiter des circonstances exceptionnelles dans un bloc `catch` et pour libérer les ressources dans le bloc `finally`.

Pour plus d'informations et des exemples sur la génération répétée d'exceptions, consultez try-catch et Génération d'exceptions. Pour plus d'informations sur le bloc `finally`, consultez try-finally.

Exemple

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

spécification du langage C#

Pour plus d'informations, consultez la section Instruction try de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Instructions try, throw et catch (C++)
- throw
- Comment : lever explicitement des exceptions
- using, instruction

Checked et Unchecked (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Les instructions C# peuvent s'exécuter dans un contexte vérifié (checked) ou non vérifié (unchecked). Dans un contexte vérifié, un dépassement de capacité arithmétique lève une exception. Dans un contexte non vérifié (unchecked), ce dépassement de capacité arithmétique est ignoré et le résultat est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire.

- checked Indique un contexte vérifié.
- unchecked Indique un contexte non vérifié.

Les opérations suivantes sont concernées par la vérification du dépassement de capacité :

- Expressions utilisant les opérateurs prédéfinis suivants dans des types intégraux :

  `++`, `--`, unaire `-`, `+`, `-`, `*`, `/`

- Conversions numériques explicites entre types intégraux, ou de `float` ou `double` en un type intégral.

Si ni `checked` ni `unchecked` ne sont spécifiés, le contexte par défaut pour les expressions non constantes (expressions évaluées au moment de l'exécution) est défini par la valeur de l'option -checked compilateur. Par défaut, la valeur de cette option n'est pas définie et les opérations arithmétiques sont exécutées dans un contexte non vérifié (unchecked).

Pour les expressions constantes (expressions pouvant être complètement évaluées au moment de la compilation), le contexte par défaut est toujours vérifié (checked). Sauf si une expression constante est placée explicitement dans un contexte non vérifié (unchecked), les dépassements de capacité qui se produisent pendant l'évaluation lors de la compilation de l'expression entraînent des erreurs au moment de la compilation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Mots clés d'instructions

checked (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le mot clé `checked` permet d'activer explicitement le contrôle de dépassement de capacité pour les conversions et les opérations arithmétiques de type intégral.

Par défaut, une expression qui contient uniquement des valeurs constantes provoque une erreur du compilateur si l'expression produit une valeur située en dehors de la plage du type de destination. Si l'expression contient une ou plusieurs valeurs non constantes, le compilateur ne détecte pas le dépassement de capacité. L'évaluation de l'expression assignée à `i2` dans l'exemple suivant ne provoque pas d'erreur du compilateur.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

Par défaut, ces expressions non constantes ne font pas l'objet d'une vérification d'un dépassement de capacité au moment de l'exécution et ne lèvent aucune exception de dépassement de capacité. L'exemple précédent affiche -2 147 483 639 comme somme de deux entiers positifs.

Le contrôle de dépassement de capacité peut être activé par les options du compilateur, la configuration de l'environnement ou l'utilisation du mot clé `checked`. Les exemples suivants montrent comment utiliser une expression `checked` ou un bloc `checked` pour détecter le dépassement de capacité produit par la somme précédente au moment de l'exécution. Les deux exemples lèvent une exception de dépassement de capacité.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

Vous pouvez utiliser le mot clé unchecked pour empêcher la vérification du dépassement de capacité.

Exemple

Cet exemple montre comment utiliser `checked` pour activer la vérification du dépassement de capacité au moment de l'exécution.

```
class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT:  " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT:  " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                          CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                          UncheckedMethod());
    }
    /*
   Output:
   CHECKED and CAUGHT:  System.OverflowException: Arithmetic operation resulted
   in an overflow.
      at ConsoleApplication1.OverFlowTest.CheckedMethod()

   CHECKED output value is: 0
   UNCHECKED output value is: -2147483639
 */
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Checked et unchecked
- unchecked

unchecked (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le mot clé `unchecked` permet de supprimer le contrôle de dépassement de capacité pour les conversions et les opérations arithmétiques de type intégral.

Dans un contexte non vérifié (unchecked), si une expression produit une valeur située hors de la plage du type de destination, le dépassement de capacité n'est pas signalé. Par exemple, comme le calcul dans l'exemple suivant s'effectue dans un bloc ou une expression `unchecked`, le fait que le résultat est trop grand pour un entier est ignoré et `int1` se voit assigner la valeur -2,147,483,639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Si l'environnement `unchecked` est supprimé, une erreur de compilation se produit. Le dépassement de capacité peut être détecté au moment de la compilation, car tous les termes de l'expression sont des constantes.

Les expressions qui contiennent des termes non constants sont non vérifiées par défaut au moment de la compilation et de l'exécution. Consultez checked pour plus d'informations sur l'activation d'un environnement vérifié (checked).

Étant donné que la vérification de dépassement de capacité prend du temps, l'utilisation de code non vérifié dans les situations où il n'existe aucun danger de dépassement de capacité peut améliorer les performances. Toutefois, si un dépassement de capacité est possible, il convient d'utiliser un environnement vérifié (checked).

Exemple

L'exemple de code suivant montre comment utiliser le mot clé `unchecked`.

```
class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de

référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Checked et unchecked
- désactivé

fixed, instruction (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

L'instruction `fixed` empêche le récupérateur de mémoire de déplacer une variable mobile. L'instruction `fixed` est uniquement autorisée dans un contexte unsafe. Vous pouvez également utiliser le mot clé `fixed` pour créer des mémoires tampons de taille fixe.

L'instruction `fixed` définit un pointeur vers une variable managée et épingle cette variable pendant l'exécution de l'instruction. Les pointeurs vers des variables managées ne sont utiles que dans un contexte `fixed`. Sans contexte `fixed`, le garbage collection peut déplacer les variables de manière imprévisible. Le compilateur C# permet seulement d'assigner un pointeur à une variable managée dans une instruction `fixed`.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

Vous pouvez initialiser un pointeur à l'aide d'un tableau, d'une chaîne, d'un tampon de taille fixe ou de l'adresse d'une variable. L'exemple suivant montre l'utilisation des adresses de variables, des tableaux et des chaînes :

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

À compter de C# 7.3, l'instruction `fixed` s'applique à d'autres types au-delà des tableaux, chaînes, mémoires tampons de taille fixe ou variables non managées. Tout type qui implémente une méthode nommée `GetPinnableReference` peut être épinglé. `GetPinnableReference` doit retourner une variable `ref` d'un type non managé. Les types .NET System.Span<T> et System.ReadOnlySpan<T> introduits dans .NET Core 2.0 utilisent ce modèle et peuvent être épinglés. Ceci est illustré dans l'exemple suivant :

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
                  1,
                1,  1,
              1,  2,  1,
            1,  3,  3,  1,
          1,  4,  6,  4,  1,
        1,  5,  10, 10, 5,  1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

Si vous créez des types qui doivent être inclus dans ce modèle, consultez Span<T>.GetPinnableReference() pour obtenir un exemple d'implémentation du modèle.

Plusieurs pointeurs peuvent être initialisés dans une instruction s'ils sont tous du même type :

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

Pour initialiser des pointeurs de types différents, imbriquez des instructions `fixed`, comme indiqué dans l'exemple suivant.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

Une fois que le code de l'instruction est exécuté, toutes les variables épinglées sont libérées et soumises au garbage collection. Par conséquent, ne pointez pas vers ces variables en dehors de l'instruction `fixed`. La portée des variables déclarées dans l'instruction `fixed` est limitée à cette instruction, ce qui facilite les choses :

```
fixed (byte* ps = srcarray, pd = dstarray)
{
   ...
}
// ps and pd are no longer in scope here.
```

Les pointeurs initialisés dans des instructions `fixed` sont des variables en lecture seule. Pour modifier la valeur du pointeur, vous devez déclarer une deuxième variable de pointeur et la modifier. La variable déclarée dans l'instruction `fixed` ne peut pas être modifiée :

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

Vous pouvez allouer de la mémoire à la pile, où elle n'est pas soumise au nettoyage de la mémoire et n'a donc pas besoin d'être épinglée. Pour ce faire, utilisez une `stackalloc` expression.

spécification du langage C#

Pour plus d'informations, consultez la section Instructions fixes de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- unsafe
- Types de pointeur
- Mémoires tampons de taille fixe

instruction lock (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

L'instruction `lock` obtient le verrou d'exclusion mutuelle d'un objet donné, exécute un bloc d'instructions, puis libère le verrou. Tant qu'un verrou est maintenu, le thread qui contient le verrou peut à nouveau obtenir et libérer le verrou. Tout autre thread se voit bloquer l'obtention du verrou et attend que ce dernier soit libéré.

L'instruction `lock` se présente sous la forme

```
lock (x)
{
    // Your code...
}
```

où `x` est une expression de type référence. Elle équivaut précisément à

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Dans la mesure où le code utilise un bloc try...finally, le verrou est libéré même si une exception est levée dans le corps d'une instruction `lock`.

Vous ne pouvez pas utiliser l'opérateur await dans le corps d'une instruction `lock`.

Recommandations

Quand vous synchronisez l'accès des threads à une ressource partagée, verrouillez une instance d'objet dédiée (par exemple `private readonly object balanceLock = new object();`) ou toute autre instance peu susceptible d'être utilisée comme objet de verrouillage par des parties du code non associées. Évitez d'utiliser la même instance d'objet de verrouillage pour différentes ressources partagées, car cela peut entraîner une contention d'interblocage ou de verrouillage. En particulier, évitez d'utiliser les éléments suivants en tant qu'objets de verrouillage :

- `this`, qui peut être utilisé en tant que verrou par les appelants.
- Les instances de Type, qui peuvent être obtenues par l'opérateur typeof ou par réflexion.
- Les instances de chaîne, notamment les littéraux de chaîne, qui peuvent être internés.

Maintenez un verrou aussi rapide que possible pour réduire la contention de verrouillage.

Exemple

L'exemple suivant définit une classe `Account`, qui synchronise l'accès à son champ `balance` privé en verrouillant une instance `balanceLock` dédiée. L'utilisation de la même instance pour le verrouillage permet de garantir que

le champ `balance` ne peut pas être mis à jour simultanément par deux threads qui tentent d'appeler les méthodes `Debit` ou `Credit` en même temps.

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
```

```
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la section Instruction lock de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- Vue d'ensemble des primitives de synchronisation

Paramètres de méthode (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Cette valeur peut être modifiée dans la méthode, mais la valeur modifiée n'est pas conservée quand le contrôle repasse à la procédure appelante. En utilisant un mot clé de paramètre de méthode, vous pouvez modifier ce comportement.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

params (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Avec le mot clé `params`, vous pouvez spécifier un paramètre de méthode qui prend un nombre variable d'arguments. Le type de paramètre doit être un tableau unidimensionnel.

Dans une déclaration de méthode, vous ne pouvez pas spécifier de paramètre supplémentaire après le mot clé `params` et vous pouvez utiliser un seul mot clé `params`.

Si le type déclaré du `params` paramètre n'est pas un tableau unidimensionnel, l'erreur de compilateur CS0225 se produit.

Lorsque vous appelez une méthode avec un `params` paramètre, vous pouvez passer :

- Liste séparée par des virgules des arguments du type des éléments du tableau.
- Tableau d'arguments du type spécifié.
- Aucun argument. Si vous n'envoyez aucun argument, la longueur de la liste `params` est égale à zéro.

Exemple

L'exemple suivant montre différentes façons d'envoyer des arguments à un paramètre `params`.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Paramètres de méthodes

in, modificateur de paramètre (référence C#)

02/11/2020 • 10 minutes to read • Edit Online

Le mot clé `in` entraîne le passage des arguments par référence. Il fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument. Il est similaire aux mots clés ref ou out, sauf que les arguments `in` ne peuvent pas être modifiés par la méthode appelée. Alors que la modification des arguments `ref` est facultative, les arguments `out` doivent être modifiés par la méthode appelée ; ces modifications sont observables dans le contexte d'appel.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

L'exemple précédent montre que le modificateur `in` n'est généralement pas nécessaire sur le site d'appel. Il l'est uniquement dans la déclaration de méthode.

> **NOTE**
>
> Le mot clé `in` peut également être utilisé avec un paramètre de type générique pour spécifier que le paramètre de type est contravariant, dans le cadre d'une instruction `foreach` ou d'une clause `join` dans une requête LINQ. Pour plus d'informations sur l'utilisation du mot clé `in` dans ces contextes, consultez in, qui fournit des liens vers toutes ces utilisations.

Les variables passées comme des arguments `in` doivent être initialisées avant d'être passées dans un appel de méthode. Toutefois, la méthode appelée ne peut pas attribuer de valeur ou modifier l'argument.

Le modificateur de paramètre `in` est disponible dans C# 7.2 et versions ultérieures. Les versions précédentes génèrent une erreur de compilateur `CS8107` (« La fonctionnalité "références en lecture seule" n'est pas disponible dans C# 7.0. Utilisez la version 7.2 ou supérieure du langage. » Pour configurer la version du langage du compilateur, consultez Sélectionner la version du langage C#.

Les mots clés `in`, `ref` et `out` ne sont pas considérés comme faisant partie de la signature de méthode à des fins de résolution de surcharge. Par conséquent, les méthodes ne peuvent pas être surchargées si la seule différence est que l'une d'elles accepte un argument `ref` ou `in` et que l'autre accepte un argument `out`. Le code suivant, par exemple, ne se compilera pas :

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Une surcharge basée sur la présence de `in` est autorisée :

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Règles de résolution de surcharge

Vous pouvez comprendre les règles de résolution de surcharge pour les méthodes par valeur par rapport à celles qui utilisent des arguments `in` en comprenant pourquoi on utilise des arguments `in`. Vous pouvez optimiser les performances en définissant des méthodes à l'aide de paramètres `in`. Certains arguments de type `struct` peuvent être volumineux et, quand les méthodes sont appelées dans des boucles serrées ou des chemins de code critiques, le coût de la copie de ces structures est crucial. Les méthodes déclarent des paramètres `in` pour spécifier que des arguments peuvent être passés par référence en toute sécurité, car la méthode appelée ne modifie pas l'état de l'argument. Le passage de ces arguments par référence permet d'éviter une copie (potentiellement) coûteuse.

La spécification de `in` pour des arguments au niveau de l'appel de site est généralement facultative. Il n'existe aucune différence sémantique entre le passage d'arguments par valeur et leur passage par référence à l'aide du modificateur `in`. Le modificateur `in` sur le site d'appel est facultatif, car vous n'avez pas besoin d'indiquer que la valeur de l'argument peut être changée. Vous ajoutez explicitement le modificateur `in` sur le site d'appel pour vérifier que l'argument est passé par référence, non par valeur. L'utilisation explicite de `in` a les deux effets suivants :

Tout d'abord, la spécification de `in` sur le site d'appel force le compilateur à sélectionner une méthode définie avec un paramètre `in` correspondant. Sinon, quand deux méthodes diffèrent uniquement par la présence de `in`, la surcharge par valeur convient mieux.

Ensuite, la spécification de `in` déclare votre intention de passer un argument par référence. L'argument utilisé avec `in` doit représenter un emplacement directement référençable. Les mêmes règles générales pour les arguments `out` et `ref` s'appliquent : vous ne pouvez pas utiliser de constantes, de propriétés ordinaires ou d'autres expressions qui produisent des valeurs. Sinon, l'omission de `in` sur le site d'appel informe le compilateur que vous l'autorisez à créer une variable temporaire à passer par référence en lecture seule à la méthode. Le compilateur crée une variable temporaire pour surmonter plusieurs restrictions avec les arguments `in` :

- Une variable temporaire autorise des constantes au moment de la compilation comme paramètres `in`.
- Une variable temporaire autorise des propriétés ou d'autres expressions pour les paramètres `in`.
- Une variable temporaire autorise des arguments pour lesquels il existe une conversion implicite de type d'argument en type de paramètre.

Dans toutes les instances précédentes, le compilateur crée une variable temporaire qui stocke la valeur de la constante, la propriété ou une autre expression.

Le code suivant illustre ces règles :

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Supposons à présent qu'une autre méthode utilisant des arguments par valeur est disponible. Les résultats changent comme illustré dans le code suivant :

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

Le seul appel de méthode dans lequel l'argument est passé par référence est le dernier.

> **NOTE**
> Le code précédent utilise `int` comme type d'argument par souci de simplicité. Comme `int` n'est pas plus volumineux qu'une référence dans la plupart des ordinateurs modernes, il n'y aucun avantage à passer un seul `int` comme référence en lecture seule.

Limitations sur les paramètres `in`

Vous ne pouvez pas utiliser les mots clés `in`, `ref` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.
- Le premier argument d'une méthode d'extension ne peut pas avoir le `in` modificateur, sauf si cet argument est un struct.
- Le premier argument d'une méthode d'extension où cet argument est un type générique (même si ce type est imposé comme struct).

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Paramètres de méthodes
- Écrire du code efficace sécurisé

ref (référence C#)

02/11/2020 • 16 minutes to read • Edit Online

Le mot clé `ref` indique une valeur qui est passée par référence. Il est utilisé dans quatre contextes différents :

- Dans une signature de méthode et dans un appel de méthode, pour passer un argument à une méthode par référence. Pour plus d'informations, consultez Passage d'un argument par référence.
- Dans une signature de méthode, pour retourner une valeur à l'appelant par référence. Pour plus d'informations, consultez Valeurs de retour de référence.
- Dans le corps d'un membre, pour indiquer qu'une valeur de retour de référence est stockée localement sous la forme d'une référence que l'appelant a l'intention de modifier, ou une variable locale accède généralement à une valeur par référence. Pour plus d'informations, consultez Variables locales ref.
- Dans une déclaration de `struct` pour déclarer un `ref struct` ou un `readonly ref struct`. Pour plus d'informations, consultez la section `ref` struct de l'article types de structures .

Passage d'un argument par référence

Quand il est utilisé dans la liste de paramètres d'une méthode, le mot clé `ref` indique qu'un argument est passé par référence, et non par valeur. Le mot clé `ref` fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument. Par exemple, si l'appelant passe une expression de variable locale ou une expression d'accès à un élément de tableau, et que la méthode appelée remplace l'objet auquel fait référence le paramètre ref, la variable locale de l'appelant ou l'élément de tableau fait désormais référence au nouvel objet lorsque la méthode est retournée.

> **NOTE**
> Ne confondez pas le concept de passage par référence avec celui de types de référence. Les deux concepts ne sont pas identiques. Un paramètre de méthode peut être modifié par `ref`, qu'il s'agisse d'un type valeur ou d'un type référence. Il n'y a aucun boxing d'un type valeur lorsqu'il est passé par référence.

Pour utiliser un paramètre `ref`, la définition de la méthode et la méthode d'appel doivent utiliser explicitement le mot clé `ref`, comme indiqué dans l'exemple suivant.

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Un argument passé à un paramètre `ref` ou `in` doit être initialisé avant d'être transmis. Cette obligation diffère des paramètres out, dont les arguments ne doivent pas être explicitement initialisés avant d'être passés.

Les membres d'une classe ne peuvent pas avoir de signatures qui diffèrent uniquement par `ref`, `in` ou `out`. Une erreur de compilation se produit si la seule différence entre deux membres d'un type est que l'un d'eux a un paramètre `ref` et que l'autre un a un paramètre `out` ou `in`. Le code suivant, par exemple, ne se compile pas.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Toutefois, les méthodes peuvent être surchargées quand une méthode a un paramètre `ref`, `in` ou `out` et que l'autre a un paramètre de valeur, comme illustré dans l'exemple suivant.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

Dans d'autres situations nécessitant une correspondance de signature, comme le masquage ou la substitution, `in`, `ref` et `out` font partie de la signature et ne correspondent pas l'un à l'autre.

Les propriétés ne sont pas des variables. Ce sont des méthodes et ne peuvent pas être passées aux paramètres `ref`.

Vous ne pouvez pas utiliser les mots clés `ref`, `in` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.

En outre, les méthodes d'extension présentent les restrictions suivantes :

- Le `out` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension.
- Le `ref` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension lorsque l'argument n'est pas un struct, ou un type générique qui n'est pas imposé comme struct.
- Le `in` mot clé ne peut pas être utilisé, sauf si le premier argument est un struct. Le `in` mot clé ne peut pas être utilisé sur un type générique, même lorsqu'il est imposé comme struct.

Passage d'un argument par référence : exemple

Les exemples précédents passent les types valeur par référence. Vous pouvez également utiliser le mot clé `ref` pour passer les types référence par référence. Le passage d'un type référence par référence permet à la méthode appelée de remplacer l'objet auquel fait référence le paramètre de référence dans l'appelant. L'emplacement de stockage de l'objet est passé à la méthode comme valeur du paramètre de référence. Si vous modifiez la valeur de l'emplacement de stockage du paramètre (pour pointer vers un nouvel objet), vous modifiez également l'emplacement de stockage auquel fait référence l'appelant. L'exemple suivant passe une instance d'un type référence en tant que paramètre `ref`.

```
class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main.  Name: Fasteners, ID: 54321
// Back in Main.  Name: Stapler, ID: 12345
```

Pour plus d'informations sur la manière de passer des types référence par valeur et par référence, consultez Passage de paramètres de type référence .

Valeurs de retour de référence

Les valeurs de retour de référence (ou retours ref) sont des valeurs qu'une méthode retourne par référence à l'appelant. Autrement dit, l'appelant peut modifier la valeur retournée par une méthode, et cette modification est reflétée dans l'état de l'objet dans la méthode d'appel.

Une valeur de retour de référence est définie à l'aide du mot clé `ref` :

- Dans la signature de la méthode. Par exemple, la signature de méthode suivante indique que la méthode `GetCurrentPrice` retourne une valeur Decimal par référence.

```
public ref decimal GetCurrentPrice()
```

- Entre le jeton `return` et la variable retournée dans une instruction `return` dans la méthode. Par exemple :

```
return ref DecimalArray[0];
```

Pour que l'appelant puisse modifier l'état de l'objet, la valeur de retour de référence doit être stockée dans une variable qui est explicitement définie comme variable locale ref.

Voici un exemple plus complet de retour de référence, qui illustre la signature de la méthode et le corps de la méthode.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

La méthode appelée peut également déclarer la valeur renvoyée en tant que `ref readonly` pour renvoyer la valeur par référence et faire en sorte que le code d'appel ne puisse pas modifier la valeur renvoyée. La méthode d'appel peut éviter de copier la valeur retournée en stockant la valeur dans une variable de référence locale en lecture seule .

Pour obtenir un exemple, consultez un exemple de retours REF et de variables locales Ref.

Variables locales ref

Une variable locale ref est utilisée pour faire référence aux valeurs retournées à l'aide de `return ref`. Vous ne pouvez pas initialiser une variable locale ref en valeur de retour non ref. En d'autres termes, la partie droite de l'initialisation doit être une référence. Toute modification apportée à la valeur de la variable locale ref est reflétée dans l'état de l'objet dont la méthode a retourné la valeur par référence.

Vous définissez une variable locale ref à l'aide du mot clé `ref` avant la déclaration de la variable, ainsi qu'immédiatement avant l'appel à la méthode qui retourne la valeur par référence.

Par exemple, l'instruction suivante définit une valeur de variable locale ref qui est retournée par une méthode nommée `GetEstimatedValue` :

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

Vous pouvez accéder à une valeur par référence de la même façon. Dans certains cas, l'accès à une valeur par référence augmente les performances en évitant une opération de copie potentiellement coûteuse. Par exemple, l'instruction suivante montre comment il est possible de définir une valeur locale ref qui est utilisée pour référencer une valeur.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

Dans les deux exemples `ref` , le mot clé doit être utilisé aux deux emplacements, ou le compilateur génère l'erreur CS8172, « impossible d'initialiser une variable par référence avec une valeur ».

À partir de C# 7.3, la variable d'itération de l'instruction `foreach` peut être une variable locale ref ou une variable locale ref readonly. Pour plus d'informations, voir l'article Instruction foreach.

À compter de C# 7,3, vous pouvez également réassigner une variable locale Ref locale ou ref ReadOnly avec l' opérateur d'assignation de référence.

Variable locale ref readonly

Une variable locale ref readonly est utilisée pour se référer à des valeurs renvoyées par la méthode ou par la propriété comprenant `ref readonly` dans sa signature et utilise `return ref`. Une variable `ref readonly` combine les propriétés d'une variable locale `ref` avec une variable `readonly` : c'est un alias du stockage auquel elle est assignée et qui ne peut pas être modifié.

Exemple de retours ref et de variables locales ref

L'exemple suivant définit une classe `Book` qui a deux champs String, `Title` et `Author`. Il définit également une classe `BookCollection` qui inclut un tableau privé d'objets `Book`. Des objets livres individuels sont retournés par référence en appelant sa méthode `GetBookByTitle`.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                             new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
                            };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

Quand l'appelant stocke la valeur retournée par la méthode `GetBookByTitle` comme variable locale ref, les modifications apportées par l'appelant à la valeur de retour sont reflétées dans l'objet `BookCollection`, comme le montre l'exemple suivant.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Écrire du code efficace sécurisé
- Retours ref et variables locales ref
- Expression ref conditionnelle
- Passer des paramètres
- Paramètres de méthodes
- Référence C#
- Guide de programmation C#
- Mots clés C#

out, modificateur de paramètre (référence C#)

02/11/2020 • 7 minutes to read • Edit Online

Le mot clé `out` entraîne le passage des arguments par référence. Il fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument. Il est similaire au mot clé ref, à la différence que `ref` nécessite que la variable soit initialisée avant d'être passée. Il est également similaire au mot clé in, à la différence que `in` n'autorise pas la méthode appelée à modifier la valeur d'argument. Pour utiliser un paramètre `out`, la définition de la méthode et la méthode d'appel doivent utiliser explicitement le mot clé `out`. Par exemple :

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

Le mot clé `out` peut également être utilisé avec un paramètre de type générique pour spécifier que le paramètre de type est covariant. Pour plus d'informations sur l'utilisation du mot clé `out` dans ce contexte, consultez out, modificateur générique.

Les variables passées comme arguments `out` n'ont pas à être initialisées avant d'être passées dans un appel de méthode. Toutefois, la méthode appelée doit assigner une valeur avant le retour de la méthode.

Les mots clés `in`, `ref` et `out` ne sont pas considérés comme faisant partie de la signature de méthode à des fins de résolution de surcharge. Par conséquent, les méthodes ne peuvent pas être surchargées si la seule différence est que l'une d'elles accepte un argument `ref` ou `in` et que l'autre accepte un argument `out`. Le code suivant, par exemple, ne se compilera pas :

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Toutefois, la surcharge est autorisée si une méthode prend un argument `ref`, `in` ou `out`, et que l'autre méthode n'a aucun de ces modificateurs, comme suit :

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

Le compilateur choisit la meilleure surcharge en mettant en correspondance les modificateurs de paramètre sur le site d'appel et les modificateurs de paramètre utilisés dans l'appel de méthode.

Les propriétés ne sont pas des variables et, par conséquent, ne peuvent pas être passées en tant que paramètres `out`.

Vous ne pouvez pas utiliser les mots clés `in`, `ref` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.

En outre, les méthodes d'extension présentent les restrictions suivantes :

- Le `out` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension.
- Le `ref` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension lorsque l'argument n'est pas un struct, ou un type générique qui n'est pas imposé comme struct.
- Le `in` mot clé ne peut pas être utilisé, sauf si le premier argument est un struct. Le `in` mot clé ne peut pas être utilisé sur un type générique, même lorsqu'il est imposé comme struct.

Déclarer des paramètres `out`

Il arrive souvent que l'on déclare une méthode avec des arguments `out` pour qu'elle retourne plusieurs valeurs. À compter de C# 7,0, considérez les tuples de valeur pour les scénarios similaires. L'exemple suivant utilise `out` pour retourner trois variables avec un seul appel de méthode. Le troisième argument est assigné à la valeur null. Cela permet aux méthodes de retourner des valeurs le cas échéant.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Appel d'une méthode avec un argument `out`

Dans C# 6 et les versions antérieures, vous devez déclarer une variable dans une instruction distincte

avant de la passer comme argument `out`. L'exemple suivant déclare une variable nommée `number` avant de la passer à la méthode Int32.TryParse, qui tente de convertir une chaîne en nombre.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

À compter de C# 7.0, vous pouvez déclarer la variable `out` dans la liste d'arguments de l'appel de méthode au lieu de le faire dans une déclaration de variable distincte. Cette technique rend le code plus simple et plus lisible, et vous empêche également d'assigner par inadvertance une valeur à la variable avant l'appel de méthode. L'exemple suivant est semblable au précédent, sauf qu'il définit la variable `number` dans l'appel de la méthode Int32.TryParse.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Dans l'exemple précédent, la variable `number` est fortement typée en `int`. Vous pouvez également déclarer une variable locale implicitement typée, comme dans l'exemple suivant.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Paramètres de méthodes

namespace (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `namespace` est utilisé pour déclarer une portée qui contient un ensemble d'objets connexes. Vous pouvez utiliser un espace de noms pour organiser les éléments de code et créer des types globaux uniques.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Remarques

Dans un espace de noms, vous pouvez déclarer zéro ou plusieurs des types suivants :

- autre espace de noms
- class
- interface
- modélis
- variables
- delegate

Le compilateur ajoute un espace de noms par défaut, que vous déclariez ou non explicitement un espace de noms dans un fichier source C#. Cet espace de noms sans nom, parfois appelé espace de noms global, est présent dans chaque fichier. Tout identificateur dans l'espace de noms global peut être utilisé dans un espace de noms nommé.

Les espaces de noms ont implicitement un accès public, et cela n'est pas modifiable. Consultez Modificateurs d'accès pour connaître les modificateurs d'accès que vous pouvez assigner à des éléments dans un espace de noms.

Il est possible de définir un espace de noms dans deux déclarations ou plus. Par exemple, l'exemple suivant définit deux classes comme appartenant à l'espace de noms `MyCompany` :

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

Exemple

L'exemple suivant montre comment appeler une méthode statique dans un espace de noms imbriqué.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

spécification du langage C#

Pour plus d'informations, voir la section Espace de noms de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- à
- using static
- Qualificateur d'alias d'espace de noms `::`
- Espaces de noms

using (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

Le mot clé `using` a trois utilisations principales :

- L'instruction using définit une portée à la fin de laquelle un objet est supprimé.
- La directive using crée un alias pour un espace de noms ou importe des types définis dans d'autres espaces de noms.
- La directive using static importe les membres d'une seule classe.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Espaces de noms
- extern

using, directive (référence C#)

16/12/2020 • 5 minutes to read • Edit Online

La directive `using` a trois utilisations :

- Pour autoriser l'utilisation de types dans un espace de noms pour ne pas avoir à qualifier l'utilisation d'un type dans cet espace de noms :

  ```
  using System.Text;
  ```

- Pour vous permettre d'accéder aux membres statiques et aux types imbriqués d'un type sans devoir qualifier l'accès avec le nom du type.

  ```
  using static System.Math;
  ```

  Pour plus d'informations, consultez la directive using static.

- Pour créer un alias pour un espace de noms ou un type. Cela s'appelle une *directive using alias*.

  ```
  using Project = PC.MyCompany.Project;
  ```

Le mot clé `using` est également utilisé pour créer des *instructions using*IDisposable*, qui garantissent que les objets* tels que les fichiers et les polices sont gérés correctement. Pour plus d'informations, consultez Instruction using.

Utilisation du type statique

Vous pouvez accéder aux membres statiques d'un type sans devoir qualifier l'accès avec le nom du type :

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Remarques

La portée d'une directive `using` se limite au fichier dans lequel elle apparaît.

La directive `using` peut être placée :

- Au début d'un fichier de code source, avant les définitions de type ou d'espace de noms.
- Dans n'importe quel espace de noms, mais avant les espaces de noms ou types déclarés dans cet espace de noms.

Sinon, une erreur de compilateur CS1529 est générée.

Créez une directive d'alias `using` afin de faciliter la qualification d'un identificateur pour un espace de noms ou un type. Dans une directive `using`, le type ou l'espace de noms complet doit toujours être utilisé, indépendamment des directives `using` placées avant. Aucun alias `using` ne peut être utilisé dans la déclaration d'une directive `using`. Par exemple, les lignes ci-dessous génèrent une erreur de compilation :

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

Créez une directive `using` pour utiliser les types dans un espace de noms sans avoir à spécifier l'espace de noms. Une directive `using` ne vous donne pas accès à des espaces de noms imbriqués dans l'espace de noms que vous spécifiez.

Les espaces de noms sont organisés en deux catégories : définis par l'utilisateur et définis par le système. Les espaces de noms définis par l'utilisateur sont des espaces de noms définis dans votre code. Pour obtenir la liste des espaces de noms définis par le système, consultez Navigateur d'API .NET.

Exemple 1

L'exemple suivant montre comment définir et utiliser un alias `using` pour un espace de noms :

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

Une directive d'alias using ne peut pas avoir un type générique ouvert sur le côté droit. Par exemple, vous ne pouvez pas créer un alias using pour un `List<T>`, mais vous pouvez en créer un pour un `List<int>`.

Exemple 2

L'exemple suivant montre comment définir une directive `using` et un alias `using` pour une classe :

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

spécification du langage C#

Pour plus d'informations, consultez Directives using dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Utilisation d'espaces de noms
- Mots clés C#
- Espaces de noms
- using, instruction

using static, directive (référence C#)

02/11/2020 • 6 minutes to read • Edit Online

La directive `using static` désigne un type dont vous pouvez accéder aux membres statiques et aux types imbriqués sans spécifier de nom de type. Sa syntaxe est la suivante :

```
using static <fully-qualified-type-name>;
```

où *fully-qualified-type-name* est le nom du type dont les membres statiques et les types imbriqués peuvent être référencés sans spécifier de nom de type. Si vous ne fournissez pas de nom de type complet (le nom de l'espace de noms complet avec le nom du type), C# génère l'erreur du compilateur CS0246 : "Le nom du type ou de l'espace de noms 'type/espace_de_nom' est introuvable (il manque peut-être une directive using ou une référence d'assembly).

La directive `using static` s'applique à tout type ayant des membres statiques (ou des types imbriqués), même s'il a également des membres d'instance. Toutefois, les membres d'instance ne peuvent être appelés que par l'instance du type.

La directive `using static` a été introduite avec C# 6.

Notes

En général, quand vous appelez un membre statique, vous indiquez le nom du type, ainsi que le nom du membre. Entrer plusieurs fois le même nom de type pour appeler des membres du type peut produire du code détaillé et peu clair. Par exemple, la définition suivante d'une classe `Circle` référence un certain nombre de membres de la classe Math.

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * Math.PI; }
   }

   public double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}
```

En éliminant la nécessité de référencer explicitement la classe Math chaque fois qu'un membre est référencé, la directive `using static` génère du code beaucoup plus clair :

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` importe uniquement les membres statiques accessibles et les types imbriqués déclarés dans le type spécifié. Les membres hérités ne sont pas importés. Vous pouvez importer à partir de n'importe quel type nommé avec une directive static, notamment des modules Visual Basic. Si des fonctions de niveau supérieur F# apparaissent dans les métadonnées comme membres statiques d'un type nommé dont le nom est un identificateur C# valide, les fonctions F# peuvent être importées.

`using static` rend les méthodes d'extension déclarées dans le type spécifié disponibles pour la recherche de méthode d'extension. Toutefois, les noms des méthodes d'extension ne sont pas importés dans la portée pour la référence non qualifiée dans le code.

Les méthodes portant le même nom et qui sont importées à partir de différents types par différentes directives `using static` dans la même unité de compilation ou le même espace de noms forment un groupe de méthodes. La résolution de surcharge au sein de ces groupes de méthodes suit des règles C# normales.

Exemple

L'exemple suivant utilise la directive `using static` pour que les membres statiques des classes Console, Math et String soient disponibles sans que vous ayez à spécifier leur nom de type.

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

Dans cet exemple, la directive `using static` aurait également pu être appliquée au type Double. Cela aurait permis d'appeler la méthode TryParse(String, Double) sans spécifier un nom de type. Toutefois, cela crée un code moins lisible, puisqu'il est nécessaire de vérifier les `using static` directives pour déterminer la méthode du type numérique qui `TryParse` est appelée.

Voir aussi

- using, directive
- Référence C#
- Mots clés C#
- Utilisation d'espaces de noms
- Espaces de noms

using, instruction (référence C#)

16/12/2020 • 7 minutes to read • Edit Online

Fournit une syntaxe pratique qui garantit l'utilisation correcte d'objets IDisposable. À compter de C# 8,0, l' `using` instruction garantit l'utilisation correcte des IAsyncDisposable objets.

Exemple

L'exemple suivant montre comment utiliser l'instruction `using`.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

À compter de C# 8,0, vous pouvez utiliser la syntaxe alternative suivante pour l' `using` instruction qui ne requiert pas d'accolades :

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do {
    item = reader.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

Remarques

File et Font sont des exemples de types managés qui accèdent à des ressources non managées (dans le cas présent, des handles de fichiers et des contextes d'appareil). Beaucoup d'autres types de ressources non managées et de bibliothèques de classes peuvent les encapsuler. Tous ces types doivent implémenter l' IDisposable interface ou l' IAsyncDisposable interface.

Quand la durée de vie d'un objet `IDisposable` est limitée à une seule méthode, vous devez le déclarer et l'instancier dans l'instruction `using`. L'instruction `using` appelle la méthode Dispose correctement sur l'objet et, quand vous l'utilisez comme indiqué précédemment, elle met également l'objet lui-même hors de portée dès que Dispose est appelée. Dans le `using` bloc, l'objet est en lecture seule et ne peut pas être modifié ou réassigné. Si l'objet implémente `IAsyncDisposable` au lieu de `IDisposable`, l' `using` instruction appelle

DisposeAsync et `awaits` le retourné ValueTask . Pour plus d'informations sur IAsyncDisposable , consultez implémenter une méthode DisposeAsync.

L' `using` instruction garantit que Dispose (ou DisposeAsync) est appelé même si une exception se produit dans le `using` bloc. Vous pouvez obtenir le même résultat en plaçant l'objet à l'intérieur d'un `try` bloc, puis Dispose en appelant (ou DisposeAsync) dans un `finally` bloc. en fait, il s'agit de la façon dont l' `using` instruction est traduite par le compilateur. L'exemple de code précédent se développe pour donner le code suivant au moment de la compilation (notez les accolades supplémentaires pour créer la portée limitée de l'objet) :

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try {
        string? item;
        do {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while(item != null);
    } finally
    {
        reader?.Dispose();
    }
}
```

La `using` syntaxe d'instruction la plus récente traduit en code similaire. Le `try` bloc s'ouvre à l'emplacement où la variable est déclarée. Le `finally` bloc est ajouté à la fermeture du bloc englobant, en général à la fin d'une méthode.

Pour plus d'informations sur l' `try` - `finally` instruction, consultez l'article try-finally .

Plusieurs instances d'un type peuvent être déclarées dans une même `using` instruction, comme illustré dans l'exemple suivant. Notez que vous ne pouvez pas utiliser de variables implicitement typées (`var`) lorsque vous déclarez plusieurs variables dans une même instruction :

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
    right = new StringReader(letters))
{
    string? item;
    do {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

Vous pouvez combiner plusieurs déclarations du même type à l'aide de la nouvelle syntaxe introduite avec C# 8, comme indiqué dans l'exemple suivant :

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using StringReader left = new StringReader(numbers),
    right = new StringReader(letters);
string? item;
do {
    item = left.ReadLine();
    Console.Write(item);
    Console.Write("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

Vous pouvez instancier l'objet de ressource, puis passer la variable à l' `using` instruction, mais ce n'est pas une meilleure pratique. Dans ce cas, une fois que le contrôle a quitté le bloc `using`, l'objet reste dans la portée mais n'a probablement pas accès à ses ressources non managées. En d'autres termes, il n'est plus complètement initialisé. Si vous essayez d'utiliser l'objet à l'extérieur du bloc `using`, vous risquez de provoquer la levée d'une exception. Pour cette raison, il est préférable d'instancier l'objet dans l' `using` instruction et de limiter son étendue au `using` bloc.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
// reader is in scope here, but has been disposed
```

Pour plus d'informations sur la suppression d'objets `IDisposable`, consultez Utilisation d'objets qui implémentent IDisposable.

spécification du langage C#

Pour plus d'informations, consultez Instruction using dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#

- Mots clés C#
- using, directive
- Garbage collection
- Utilisation d'objets implémentant IDisposable
- Interface IDisposable
- instruction using en C# 8,0

extern alias (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Il se peut que vous deviez référencer deux versions d'assemblys qui ont le même nom de type qualifié complet. Par exemple, vous pourriez avoir à utiliser plusieurs versions d'un assembly dans la même application. En utilisant un alias d'assembly externe, vous pouvez encapsuler les espaces de noms de chaque assembly dans des espaces de noms racines nommés par l'alias, ce qui permet de les utiliser dans le même fichier.

> **NOTE**
>
> Le mot clé extern est également utilisé comme modificateur de méthode, déclarant une méthode écrite en code non managé.

Pour référencer deux assemblys portant le même nom de type qualifié complet, vous devez spécifier un alias à l'invite de commandes, comme suit :

`/r:GridV1=grid.dll`

`/r:GridV2=grid20.dll`

Cela crée les alias externes `GridV1` et `GridV2`. Pour utiliser ces alias à partir d'un programme, référencez-les en utilisant le mot clé `extern`. Par exemple :

`extern alias GridV1;`

`extern alias GridV2;`

Chaque déclaration d'alias externe introduit un espace de noms racine supplémentaire qui est parallèle à l'espace de noms global (mais n'est pas compris dedans). Ainsi, vous pouvez référencer les types de chaque assembly sans ambiguïté en utilisant leur nom qualifié complet, enraciné dans l'alias d'espace de noms approprié.

Dans l'exemple précédent, `GridV1::Grid` serait le contrôle de grille de `grid.dll`, et `GridV2::Grid` serait le contrôle de grille de `grid20.dll`.

Utilisation de Visual Studio

Si vous utilisez Visual Studio, les alias peuvent être fournis de la même façon.

Ajoutez une référence de *grid.dll* et *grid20.dll* à votre projet dans Visual Studio. Ouvrez un onglet de propriété et modifiez les alias de global en GridV1 et GridV2, respectivement.

Utilisez ces alias de la même façon que ci-dessus

```
extern alias GridV1;

extern alias GridV2;
```

Vous pouvez maintenant créer un alias pour un espace de noms ou un type à *l'aide de la directive d'alias*. Pour plus d'informations, consultez using, directive.

```
using Class1V1 = GridV1::Namespace.Class1;

using Class1V2 = GridV2::Namespace.Class1;
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- ::, Opérateur
- -reference (Options du compilateur C#)

is (référence C#)

16/12/2020 • 13 minutes to read • Edit Online

L'opérateur `is` vérifie si le résultat d'une expression est compatible avec un type donné, ou (à compter de C# 7.0) teste une expression par rapport à un modèle. Pour plus d'informations sur l'opérateur de test `is` de type, consultez la section opérateur is de l'article relatif aux opérateurs de test de type et de conversion .

Utilisation des critères spéciaux avec `is`

À compter de C# 7.0, les instructions `is` et switch prennent en charge les critères spéciaux. Le mot clé `is` prend en charge les modèles suivants :

- Modèle de type, qui teste si une expression peut être convertie en un type spécifié et, si possible, effectue un cast de la variable en une variable de ce type.
- Modèle de constante : teste si une expression correspond à une valeur de constante spécifiée.
- Modèle de variable : correspondance qui réussit toujours et lie la valeur d'une expression à une variable locale.

Modèle de type

Lorsque vous utilisez le modèle de type pour rechercher des critères spéciaux, `is` permet de tester si une expression peut être convertie en un type spécifié et, si tel est le cas, effectuer un cast de l'expression en une variable de ce type. Il s'agit d'une extension simple de l'instruction `is` qui permet une évaluation et une conversion de type concises. La forme générale du modèle de type `is` est la suivante :

```
expr is type varname
```

Où *expr* est une expression qui prend la valeur d'une instance d'un certain type, *type* est le nom du type dans lequel le résultat de *expr* doit être converti, et *varname* est l'objet vers lequel le résultat de *expr* est converti si le `is` test a la valeur `true` .

L' `is` expression est `true` si *expr* n'est pas `null` et que l'une des conditions suivantes est vraie :

- *expr* est une instance du même type que *type*.
- *expr* est une instance d'un type qui dérive de *type*. En d'autres termes, le résultat de *expr* peut être upcasté en une instance de *type*.
- *expr* a un type au moment de la compilation qui est une classe de base de *type* et *expr* a un type au moment de l'exécution égal à *type* ou dérivé de *type*. Le *type au moment de la compilation* d'une variable est le type de la variable, tel qu'il est défini dans sa déclaration. Le *type au moment de l'exécution* d'une variable est le type de l'instance qui est assignée à cette variable.
- *expr* est une instance d'un type qui implémente l'interface *type*.

À compter de C# 7.1, *expr* peut avoir un type à la compilation défini par un paramètre de type générique et ses contraintes.

Si *exp* est `true` et que `is` est utilisé avec une instruction `if` , *varname* est assigné dans l'instruction `if` uniquement. La portée de *varname* provient de l'expression `is` à la fin du bloc englobant l'instruction `if`. L'utilisation de *varname* dans un autre emplacement génère une erreur de compilation pour l'utilisation d'une variable qui n'a pas été assignée.

L'exemple suivant utilise le modèle de type `is` pour fournir l'implémentation de la méthode

IComparable.CompareTo(Object) d'un type.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        if (o is Employee e)
        {
            return Name.CompareTo(e.Name);
        }
        throw new ArgumentException("o is not an Employee object.");
    }
}
```

Sans critères spéciaux, ce code peut être écrit comme suit. L'utilisation de critères spéciaux de type génère un code lisible plus compact en éliminant la nécessité de tester si le résultat d'une conversion est un `null`.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        var e = o as Employee;
        if (e == null)
        {
           throw new ArgumentException("o is not an Employee object.");
        }
        return Name.CompareTo(e.Name);
    }
}
```

Le modèle de type `is` génère également du code plus compact lorsqu'il détermine le type d'un type valeur. L'exemple suivant utilise le modèle de type `is` pour déterminer si un objet est une instance `Person` ou `Dog` avant d'afficher la valeur d'une propriété appropriée.

```
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person p) {
         Console.WriteLine(p.Name);
      }
      else if (o is Dog d) {
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
// Jane
// Alaskan Malamute
```

Le code équivalent sans critères spéciaux nécessite une attribution distincte comprenant un cast explicite.

```
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person) {
         Person p = (Person) o;
         Console.WriteLine(p.Name);
      }
      else if (o is Dog) {
         Dog d = (Dog) o;
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
//       Jane
//       Alaskan Malamute
```

Modèle de constante

Lorsque vous utilisez des critères spéciaux avec le modèle de constante, `is` teste si une expression est égale à une constante spécifiée. Avec C# 6 et les versions antérieures, le modèle de constante est pris en charge par l'instruction switch. À compter de C# 7.0, il est également pris en charge par l'instruction `is`. Sa syntaxe est la suivante :

```
   expr is constant
```

où *expr* est l'expression à évaluer, et où *constant* est la valeur à tester. *constant* peut correspondre à l'une des expressions constantes suivantes :

- Une valeur littérale

- Le nom d'une variable `const` déclarée
- Une constante d'énumération

L'expression constante est évaluée de la manière suivante :

- Si *expr* et *constant* sont des types intégraux, l'opérateur d'égalité C# détermine si l'expression retourne `true` (autrement dit, si `expr == constant`).
- Sinon, la valeur de l'expression est déterminée par un appel à la méthode statique Object.Equals(expr, constant).

L'exemple suivant combine le modèle de type et le modèle de constante pour tester si un objet est une instance `Dice` et, si c'est le cas, pour déterminer si la valeur obtenue après un lancer de dés est de 6.

```
using System;

public class Dice
{
    Random rnd = new Random();
    public Dice()
    {
    }
    public int Roll()
    {
        return rnd.Next(1, 7);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var d1 = new Dice();
        ShowValue(d1);
    }

    private static void ShowValue(object o)
    {
        const int HIGH_ROLL = 6;

        if (o is Dice d && d.Roll() is HIGH_ROLL)
            Console.WriteLine($"The value is {HIGH_ROLL}!");
        else
            Console.WriteLine($"The dice roll is not a {HIGH_ROLL}!");
     }
}
// The example displays output like the following:
//      The value is 6!
```

La vérification de `null` peut être effectuée à l'aide du modèle de constante. Le mot clé `null` est pris en charge par l'instruction `is`. Sa syntaxe est la suivante :

```
   expr is null
```

L'exemple suivant illustre une comparaison des vérifications de `null` :

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object o = null;

        if (o is null)
        {
            Console.WriteLine("o does not have a value");
        }
        else
        {
            Console.WriteLine($"o is {o}");
        }

        int? x = 10;

        if (x is null)
        {
            Console.WriteLine("x does not have a value");
        }
        else
        {
            Console.WriteLine($"x is {x.Value}");
        }

        // 'null' check comparison
        Console.WriteLine($"'is' constant pattern 'null' check result : { o is null }");
        Console.WriteLine($"object.ReferenceEquals 'null' check result : { object.ReferenceEquals(o, null)
}");
        Console.WriteLine($"Equality operator (==) 'null' check result : { o == null }");
    }

    // The example displays the following output:
    // o does not have a value
    // x is 10
    // 'is' constant pattern 'null' check result : True
    // object.ReferenceEquals 'null' check result : True
    // Equality operator (==) 'null' check result : True
}
```

L'expression `x is null` est calculée différemment pour les types référence et les types valeur Nullable. Pour les types valeur Nullable, elle utilise Nullable<T>.HasValue . Pour les types référence, elle utilise `(object)x == null` .

Modèle de variable

Une correspondance de modèle avec le `var` modèle est toujours réussie. Sa syntaxe est la suivante :

```
    expr is var varname
```

Où la valeur de *expr* est toujours assignée à une variable locale nommée *varname*. *varname* est une variable du même type que le type au moment de la compilation de *expr*.

Si *expr* prend la valeur `null` , l' `is` expression produit `true` et assigne `null` à *varname*. Le modèle var est l'une des rares utilisations de `is` qui produit `true` une `null` valeur.

Vous pouvez utiliser le `var` modèle pour créer une variable temporaire dans une expression booléenne, comme le montre l'exemple suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] testSet = { 100271, 234335, 342439, 999683 };

        var primes = testSet.Where(n => Factor(n).ToList() is var factors
                                    && factors.Count == 2
                                    && factors.Contains(1)
                                    && factors.Contains(n));

        foreach (int prime in primes)
        {
            Console.WriteLine($"Found prime: {prime}");
        }
    }

    static IEnumerable<int> Factor(int number)
    {
        int max = (int)Math.Sqrt(number);
        for (int i = 1; i <= max; i++)
        {
            if (number % i == 0)
            {
                yield return i;
                if (i != number / i)
                {
                    yield return number / i;
                }
            }
        }
    }
}
// The example displays the following output:
//       Found prime: 100271
//       Found prime: 999683
```

Dans l'exemple précédent, la variable temporaire est utilisée pour stocker le résultat d'une opération coûteuse. La variable peut ensuite être utilisée plusieurs fois.

spécification du langage C#

Pour plus d'informations, consultez la section L'opérateur is de la Specification du langage C# et les propositions de langage C# suivantes :

- Critères spéciaux
- Critères spéciaux avec génériques

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Opérateurs de conversion et de test de type

new, contrainte (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

La contrainte `new` spécifie que tout argument de type dans une déclaration de classe générique doit avoir un constructeur sans paramètre public. Pour utiliser la contrainte `new`, le type ne doit pas être abstract.

Appliquez la contrainte `new` à un paramètre de type quand une classe générique crée des instances du type, comme illustré dans l'exemple suivant :

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

Si vous utilisez la contrainte `new()` avec d'autres contraintes, spécifiez-la en dernier :

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

Pour plus d'informations, consultez Contraintes sur les paramètres de type.

Vous pouvez également utiliser le mot clé `new` pour créer une instance d'un type ou l'utiliser comme un modificateur de déclaration de membre.

spécification du langage C#

Pour plus d'informations, consultez la section Contraintes de paramètre de type de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Génériques

where (contrainte de type générique) (Référence C#)

02/11/2020 • 8 minutes to read • Edit Online

La clause `where` dans une définition générique spécifie des contraintes sur les types qui sont utilisés comme arguments pour les paramètres de type d'un type générique, d'une méthode, d'un délégué ou d'une fonction locale. Les contraintes peuvent spécifier des interfaces, des classes de base ou exiger qu'un type générique soit une référence, une valeur ou un type non managé. Elles déclarent les fonctionnalités que l'argument de type doit avoir.

Vous pouvez, par exemple, déclarer une classe générique, `MyGenericClass`, de telle sorte que le paramètre de type `T` implémente l'interface IComparable<T> :

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **NOTE**
>
> Pour plus d'informations sur la clause where dans une expression de requête, consultez where, clause.

La clause `where` peut également inclure une contrainte de classe de base. La contrainte de classe de base indique qu'un type à utiliser comme argument de type pour ce type générique a la classe spécifiée comme classe de base, ou est cette classe de base. Si la contrainte de classe de base est utilisée, elle doit apparaître avant toute autre contrainte sur ce paramètre de type. Certains types ne sont pas autorisés comme contrainte de classe de base : Object, Array et ValueType. Avant C# 7,3, Enum , Delegate et MulticastDelegate étaient également interdits en tant que contraintes de classe de base. L'exemple suivant montre les types qui peuvent maintenant être spécifiés comme classe de base :

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

Dans un contexte Nullable en C# 8,0 et versions ultérieures, la possibilité de valeur null du type de classe de base est appliquée. Si la classe de base n'accepte pas les valeurs null (par exemple `Base`), l'argument de type ne doit pas avoir la valeur null. Si la classe de base accepte la valeur null (par exemple `Base?`), l'argument de type peut être un type de référence Nullable ou non Nullable. Le compilateur émet un avertissement si l'argument de type est un type référence Nullable lorsque la classe de base n'accepte pas les valeurs NULL.

La clause `where` peut spécifier que le type est une `class` ou un `struct`. La contrainte `struct` supprime la nécessité de spécifier une contrainte de classe de base de `System.ValueType`. Le type `System.ValueType` ne doit pas être utilisé comme contrainte de classe de base. L'exemple suivant montre les contraintes `class` et `struct` :

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

Dans un contexte Nullable en C# 8,0 et versions ultérieures, la `class` contrainte requiert qu'un type soit un type référence qui n'accepte pas les valeurs NULL. Pour autoriser les types référence Nullable, utilisez la `class?` contrainte, qui autorise à la fois les types référence Nullable et non Nullable.

La `where` clause peut inclure la `notnull` contrainte. La `notnull` contrainte limite le paramètre de type aux types non nullables. Ce type peut être un type valeur ou un type référence non Nullable. La `notnull` contrainte est disponible à partir de C# 8,0 pour le code compilé dans un `nullable enable` contexte. Contrairement à d'autres contraintes, si un argument de type viole la `notnull` contrainte, le compilateur génère un avertissement au lieu d'une erreur. Les avertissements sont générés uniquement dans un `nullable enable` contexte.

> **IMPORTANT**
>
> Les déclarations génériques qui incluent la `notnull` contrainte peuvent être utilisées dans un contexte oublie Nullable, mais le compilateur n'applique pas la contrainte.

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

La clause `where` peut aussi inclure une contrainte `unmanaged`. La contrainte `unmanaged` limite le paramètre de type aux types connus sous le nom de types non managés. La contrainte `unmanaged` facilite l'écriture de code interop de bas niveau en C#. Cette contrainte permet des routines réutilisables sur tous les types non managés. La contrainte `unmanaged` ne peut pas être combinée avec la contrainte `class` ou `struct`. La contrainte `unmanaged` exige que le type doit être un `struct` :

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

La clause `where` peut également inclure une contrainte de constructeur, `new()`. Cette contrainte permet de créer une instance d'un paramètre de type à l'aide de l'opérateur `new`. La contrainte New () permet au compilateur de savoir que tout argument de type fourni doit avoir un constructeur sans paramètre accessible. Par exemple :

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

La contrainte `new()` apparaît en dernier dans la clause `where`. La contrainte `new()` ne peut pas être combinée avec les contraintes `struct` ou `unmanaged`. Tous les types répondant à ces contraintes doivent avoir un constructeur sans paramètre accessible, ce qui rend la contrainte `new()` redondante.

Avec plusieurs paramètres de type, utilisez une clause `where` pour chaque paramètre de type, par exemple :

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

Vous pouvez également joindre des contraintes aux paramètres de type des méthodes génériques, comme montré dans l'exemple suivant :

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notez que la syntaxe décrivant les contraintes de paramètre de type sur les délégués est la même que celle des méthodes :

```
delegate T MyDelegate<T>() where T : new();
```

Pour plus d'informations sur les délégués génériques, consultez Délégués génériques.

Pour plus d'informations sur la syntaxe et l'utilisation de contraintes, consultez Contraintes sur les paramètres de type.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Introduction aux génériques
- nouvelle contrainte
- Contraintes sur les paramètres de type

base (référence C#)

18/03/2020 • 3 minutes to read • Edit Online

Le mot clé `base` sert à accéder aux membres de la classe de base à partir d'une classe dérivée :

- Il appelle une méthode de la classe de base qui a été substituée par une autre méthode.
- Il spécifie quel constructeur de classe de base doit être appelé lors de la création d'instances de la classe dérivée.

L'accès à une classe de base est autorisé uniquement dans un constructeur, une méthode d'instance ou un accesseur de propriété d'instance.

L'utilisation du mot clé `base` à partir d'une méthode statique est une erreur.

La classe de base accessible est la classe de base spécifiée dans la déclaration de classe. Par exemple, si vous spécifiez `class ClassB : ClassA`, les membres de ClassA sont accessibles à partir de ClassB, quelle que soit la classe de base de ClassA.

Exemple

Dans cet exemple, la classe de base `Person` et la classe dérivée `Employee` ont toutes les deux une méthode nommée `Getinfo`. En utilisant le mot clé `base`, vous pouvez appeler la méthode `Getinfo` de la classe de base à partir de la classe dérivée.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Pour obtenir d'autres exemples, consultez new, virtual et override.

Exemple

Cet exemple montre comment spécifier le constructeur de classe de base qui est appelé lors de la création d'instances d'une classe dérivée.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C
- Guide de programmation C#
- Mots clés C#
- Ce

this (référence C#)

18/03/2020 • 2 minutes to read • Edit Online

Le mot clé `this` fait référence à l'instance actuelle de la classe et est également utilisé comme modificateur du premier paramètre d'une méthode d'extension.

> **NOTE**
>
> Cet article traite de l'utilisation de `this` avec des instances de classe. Pour plus d'informations sur son utilisation dans les méthodes d'extension, consultez Méthodes d'extension.

Voici quelques utilisations courantes de `this` :

- Pour qualifier des membres masqués par des noms similaires, par exemple :

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- Pour passer un objet comme paramètre à d'autres méthodes, par exemple :

```
CalcTax(this);
```

- Pour déclarer des indexeurs, par exemple :

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Les fonctions membres statiques, car ils existent au niveau de la classe et non comme faisant partie d'un objet, n'ont pas de pointeur `this`. Une erreur consiste à faire référence à `this` dans une méthode statique.

Exemple

Dans cet exemple, `this` est utilisé pour qualifier les membres de la classe `Employee`, `name` et `alias`, qui sont masqués par des noms similaires. Il est également utilisé pour passer un objet à la méthode `CalcTax`, qui appartient à une autre classe.

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C
- Guide de programmation C#

- Mots clés C#
- base
- Méthodes

null (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `null` est un littéral qui représente une référence null, c'est-à-dire qui ne fait pas référence à un objet. `null` est la valeur par défaut des variables de type référence. Les types valeur ordinaires ne peuvent pas être null, à l'exception des types valeur Nullable.

L'exemple suivant illustre certains comportements du `null` mot clé :

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Valeurs par défaut des types C#
- Nothing (Visual Basic)

bool (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le `bool` mot clé type est un alias pour le System.Boolean type de structure .net qui représente une valeur booléenne, qui peut être `true` ou `false` .

Pour effectuer des opérations logiques avec des valeurs de `bool` type, utilisez des opérateurs logiques booléens . Le `bool` type est le type de résultat des opérateurs de comparaison et d' égalité . Une `bool` expression peut être une expression conditionnelle de contrôle dans les instructions If, do, whileet for et dans l' opérateur `?:` conditionnel .

La valeur par défaut du `bool` type est `false` .

Littéraux

Vous pouvez utiliser les `true` `false` littéraux et pour initialiser une `bool` variable ou pour passer une `bool` valeur :

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Logique booléenne à trois valeurs

Utilisez le `bool?` type Nullable, si vous avez besoin de prendre en charge la logique à trois valeurs, par exemple, lorsque vous utilisez des bases de données qui prennent en charge un type booléen à trois valeurs. Dans le cas des opérandes `bool?`, les opérateurs prédéfinis `&` et `|` prennent en charge la logique à trois valeurs. Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Pour plus d'informations sur les types valeur Nullable, consultez types valeur Nullable.

Conversions

C# fournit uniquement deux conversions qui impliquent le `bool` type. Il s'agit d'une conversion implicite vers le type Nullable correspondant `bool?` et d'une conversion explicite du `bool?` type. Toutefois, .NET fournit des méthodes supplémentaires que vous pouvez utiliser pour convertir vers ou à partir du `bool` type. Pour plus d'informations, consultez la section conversion de valeurs booléennes en et à partir de la System.Boolean page de référence des API.

spécification du langage C#

Pour plus d'informations, consultez la section type bool de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- opérateurs true et false

default (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `default` peut être utilisé de deux façons :

- Pour spécifier l'étiquette par défaut dans l' `switch` instruction.
- Comme opérateur ou littéral par défaut pour produire la valeur par défaut d'un type.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

Mots clés contextuels (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Un mot clé contextuel sert à donner une signification spécifique dans le code, sans pour autant être un mot réservé en C#. Cette section traite des mots clés contextuels suivants :

MOT CLÉ	DESCRIPTION
add	Définit un accesseur d'événement personnalisé qui est appelé quand le code client s'abonne à l'événement.
async	Indique que la méthode modifiée, une expression lambda ou une méthode anonyme sont asynchrones.
await	Interrompt une méthode async jusqu'à ce qu'une tâche attendue soit terminée.
dynamic	Définit un type référence qui permet des opérations au cours desquelles il ignore la vérification de type au moment de la compilation.
get	Définit une méthode d'accesseur pour une propriété ou un indexeur.
Généralités	Alias de l'espace de noms global, qui sinon n'a pas de nom.
partielle	Définit des classes, des structures et des interfaces partielles dans la même unité de compilation.
remove	Définit un accesseur d'événement personnalisé qui est appelé quand le code client annule son abonnement à l'événement.
set	Définit une méthode d'accesseur pour une propriété ou un indexeur.
value	Utilisé pour définir des accesseurs, et ajouter ou supprimer des gestionnaires d'événements.
var	Permet que le type d'une variable déclarée à la portée de la méthode soit déterminé par le compilateur.
à	Spécifie une condition de filtre pour un bloc `catch` ou l'étiquette `case` d'une instruction `switch`.
where	Ajoute des contraintes à une déclaration générique. (Voir aussi where.)
yield	Utilisé dans un bloc itérateur pour retourner une valeur à l'objet énumérateur ou signaler la fin de l'itération.

Tous les mots clés de requête introduits dans C# 3.0 sont également contextuels. Pour plus d'informations, consultez Mots clés de requête (LINQ).

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

add (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `add` est utilisé pour définir un accesseur d'événement personnalisé qui est appelé quand le code client s'abonne à votre événement (event). Si vous fournissez un accesseur `add` personnalisé, vous devez également fournir un accesseur remove.

Exemple

L'exemple suivant illustre un événement qui a des accesseurs `add` et remove personnalisés. Pour obtenir un exemple complet, consultez comment implémenter des événements d'interface.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

En général, vous n'avez pas besoin de fournir vos propres accesseurs d'événements personnalisés. Les accesseurs générés automatiquement par le compilateur quand vous déclarez un événement sont suffisants pour la plupart des scénarios.

Voir aussi

- Événements

get (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

Le mot clé `get` définit une méthode *Accessor* dans une propriété ou un indexeur qui retourne la valeur de la propriété ou l'élément de l'indexeur. Pour plus d'informations, consultez Propriétés, Propriétés implémentées automatiquement et Indexeurs.

L'exemple suivant définit un accesseur `get` et un accesseur `set` pour une propriété nommée `Seconds`. Il utilise un champ privé nommé `_seconds` pour stocker la valeur de la propriété.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Souvent, l'accesseur `get` se compose d'une seule instruction qui retourne une valeur, comme dans l'exemple précédent. À compter de C# 7.0, vous pouvez implémenter l'accesseur `get` comme membre expression-bodied. L'exemple suivant implémente l'accesseur `get` et l'accesseur `set` en tant que membres expression-bodied.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Pour les cas simples dans lesquels les accesseurs `get` et `set` de la propriété n'effectuent aucune autre opération que la définition ou la récupération d'une valeur dans un champ de stockage privé, vous pouvez tirer parti de la prise en charge des propriétés implémentées automatiquement fournies par le compilateur C#. L'exemple suivant implémente `Hours` en tant que propriété implémentée automatiquement.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Propriétés

partial, type (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Les définitions de type partiel permettent de fractionner en plusieurs fichiers la définition d'une classe, d'une structure, d'une interface ou d'un enregistrement.

Dans *file1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

Dans *file2.cs* , la déclaration :

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarques

Fractionner un type de classe, de struct ou d'interface entre plusieurs fichiers peut être utile si vous travaillez sur des projets volumineux ou des projets contenant du code généré automatiquement, comme celui fourni par le Concepteur Windows Forms. Un type partiel peut contenir une méthode partielle. Pour plus d'informations, consultez la page Classes et méthodes partielles.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs
- Introduction aux génériques

partial, méthode (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

La signature d'une méthode partielle est définie dans une partie d'un type partiel, tandis que son implémentation est définie dans une autre partie du type. Les méthodes partielles permettent aux concepteurs de classes de fournir des hooks de méthode, semblables aux gestionnaires d'événements, que les développeurs peuvent décider ou non d'implémenter. Si le développeur ne fournit pas d'implémentation, le compilateur supprime la signature au moment de la compilation. Les méthodes partielles obéissent aux conditions suivantes :

- Les signatures des deux parties du type partiel doivent correspondre.
- La méthode doit retourner void.
- Aucun modificateur d'accès n'est autorisé. Les méthodes partielles sont implicitement privées.

L'exemple suivant illustre une méthode partielle définie dans deux parties d'une classe partielle :

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Pour plus d'informations, consultez la page Classes et méthodes partielles.

Voir aussi

- Référence C#
- type partiel

remove (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `remove` est utilisé pour définir un accesseur d'événement personnalisé qui est appelé quand le code client annule son abonnement à votre événement (event). Si vous fournissez un accesseur `remove` personnalisé, vous devez également fournir un accesseur add.

Exemple

L'exemple suivant illustre un événement qui a des accesseurs add et `remove` personnalisés. Pour obtenir un exemple complet, consultez comment implémenter des événements d'interface.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

En général, vous n'avez pas besoin de fournir vos propres accesseurs d'événements personnalisés. Les accesseurs générés automatiquement par le compilateur quand vous déclarez un événement sont suffisants pour la plupart des scénarios.

Voir aussi

- Événements

set (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé `set` définit une méthode *accessor* dans une propriété ou un indexeur qui assigne une valeur à l'élément de la propriété ou de l'indexeur. Pour plus d'informations et des exemples, consultez Propriétés, Propriétés implémentées automatiquement et Indexeurs.

L'exemple suivant définit un accesseur `get` et un accesseur `set` pour une propriété nommée `Seconds`. Il utilise un champ privé nommé `_seconds` pour stocker la valeur de la propriété.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Souvent, l'accesseur `set` se compose d'une seule instruction qui assigne une valeur, comme dans l'exemple précédent. À compter de C# 7.0, vous pouvez implémenter l'accesseur `set` comme membre expression-bodied. L'exemple suivant implémente l'accesseur `get` et l'accesseur `set` en tant que membres expression-bodied.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Pour les cas simples dans lesquels les accesseurs `get` et `set` de la propriété n'effectuent aucune autre opération que la définition ou la récupération d'une valeur dans un champ de stockage privé, vous pouvez tirer parti de la prise en charge des propriétés implémentées automatiquement fournies par le compilateur C#. L'exemple suivant implémente `Hours` en tant que propriété implémentée automatiquement.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Propriétés

when (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Vous pouvez utiliser le `when` mot clé contextuel pour spécifier une condition de filtre dans les contextes suivants :

- dans l'instruction `catch` d'un bloc try/catch ou try/catch/finally ;
- dans l'étiquette `case` d'une instruction switch.
- Dans l' `switch` expression.

`when` dans une instruction `catch`

Depuis C# 6, `when` peut être utilisé dans une instruction `catch` pour spécifier une condition qui doit avoir la valeur True pour que le gestionnaire d'une exception spécifique soit exécuté. Sa syntaxe est la suivante :

```
catch (ExceptionType [e]) when (expr)
```

où *expr* est une expression qui prend une valeur booléenne. Si la valeur retournée est `true`, le gestionnaire de l'exception est exécuté ; si la valeur retournée est `false`, il n'est pas exécuté.

L'exemple suivant utilise le mot clé `when` afin d'exécuter conditionnellement des gestionnaires pour une exception HttpRequestException selon le texte du message d'exception.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localHost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

`when` dans une instruction `switch`

Depuis C# 7.0, les étiquettes `case` n'ont plus besoin de s'exclure mutuellement et l'ordre dans lequel les étiquettes `case` s'affichent dans une instruction `switch` peut déterminer le bloc switch exécuté. Le mot clé `when` peut être utilisé pour spécifier une condition de filtre qui fait que son étiquette case associée a la valeur True uniquement si la condition de filtre a également la valeur True. Sa syntaxe est la suivante :

```
case (expr) when (when-condition):
```

où *expr* est un modèle de constante ou de type qui est comparé à l'expression de correspondance, et *when-condition* représente toute expression booléenne.

L'exemple suivant utilise le mot clé `when` pour tester les objets `Shape` ayant une superficie égale à zéro, ainsi que pour tester divers objets `Shape` ayant une superficie supérieure à zéro.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
```

```
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                         new Rectangle(10, 10), sh, new Square(0) };
      foreach (var shape in shapes)
         ShowShapeInfo(shape);
   }

   private static void ShowShapeInfo(Object obj)
   {
      switch (obj)
      {
         case Shape shape when shape.Area == 0:
            Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
            break;
         case Square sq when sq.Area > 0:
            Console.WriteLine("Information about the square:");
            Console.WriteLine($"   Length of a side: {sq.Side}");
            Console.WriteLine($"   Area: {sq.Area}");
            break;
         case Rectangle r when r.Area > 0:
            Console.WriteLine("Information about the rectangle:");
            Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Shape shape:
            Console.WriteLine($"A {shape.GetType().Name} shape");
            break;
         case null:
            Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
            break;
         default:
            Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
            break;
      }
   }
}
```

```
// The example displays the following output:
//       Information about the square:
//          Length of a side: 10
//          Area: 100
//       Information about the rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       Information about the rectangle:
//          Dimensions: 10 x 10
//          Area: 100
//       The obj variable is uninitialized.
//       The shape: Square with no dimensions
```

Voir aussi

- Instruction switch
- try/catch, instruction
- try/catch/finally, instruction

value (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `value` est utilisé dans l' `set` accesseur dans les déclarations de propriété et d' indexeur . Elle est similaire à un paramètre d'entrée d'une méthode. Le mot `value` fait référence à la valeur que le code client tente d'assigner à la propriété ou à l'indexeur. Dans l'exemple suivant, `MyDerivedClass` a une propriété appelée `Name` qui utilise le paramètre `value` pour assigner une nouvelle chaîne au `name` du champ de stockage. Du point de vue du code client, l'opération est écrite comme une assignation simple.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

Pour plus d'informations, consultez les articles Propriétés et indexeurs .

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

yield (Référence C#)

02/11/2020 • 7 minutes to read • Edit Online

Lorsque vous utilisez le `yield` mot clé contextuel dans une instruction, vous indiquez que la méthode, l'opérateur ou l'accesseur `get` dans lequel il apparaît est un itérateur. L'utilisation de `yield` pour définir un itérateur rend une classe explicite supplémentaire inutile (la classe qui contient l'état d'une énumération ; pour obtenir un exemple, consultez IEnumerator<T>) lorsque vous implémentez les modèles IEnumerable et IEnumerator pour un type de collection personnalisé.

L'exemple suivant montre les deux formulaires de l'instruction `yield`.

```
yield return <expression>;
yield break;
```

Remarques

Utilisez une instruction `yield return` pour retourner chaque élément un par un.

La séquence retournée par une méthode d'itérateur peut être consommée à l'aide d'une instruction foreach ou d'une requête LINQ. Chaque itération de la boucle `foreach` appelle la méthode Iterator. Lorsqu'une instruction `yield return` est atteinte dans la méthode Iterator, `expression` est retourné, et l'emplacement dans le code est conservé. L'exécution redémarrera à partir de cet emplacement la prochaine fois que la fonction d'itérateur sera appelée.

Utilisez une instruction `yield break` pour terminer l'itération.

Pour plus d'informations sur les itérateurs, consultez Itérateurs.

Méthodes Iterator et accesseurs get

La déclaration d'un itérateur doit respecter les exigences suivantes :

- Le type de retour doit être IEnumerable, IEnumerable<T>, IEnumerator, ou IEnumerator<T>.
- La déclaration ne peut avoir aucun paramètre in, ref ou out.

Le type `yield` d'un itérateur qui retourne IEnumerable ou IEnumerator est `object`. Si l'itérateur retourne IEnumerable<T> ou IEnumerator<T>, il doit exister une conversion implicite du type de l'expression dans l'instruction `yield return` au paramètre de type générique.

Vous ne pouvez pas inclure une instruction `yield return` ou `yield break` dans :

- Expressions lambda et méthodes anonymes.
- Méthodes qui contiennent des blocs unsafe. Pour plus d'informations, consultez unsafe.

Gestion des exceptions

Il ne peut pas y avoir d'instruction `yield return` dans un bloc try-catch. Une instruction `yield return` peut se trouver dans le bloc try d'une instruction try-finally.

Une instruction `yield break` peut se trouver dans un bloc try ou un bloc catch mais pas dans un bloc finally.

Si le corps `foreach` (en dehors de la méthode Iterator) lève une exception, un bloc `finally` de la méthode Iterator est exécuté.

Implémentation technique

Le code suivant retourne `IEnumerable<string>` depuis une méthode Iterator, puis itère au sein de ses éléments.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
   ...
}
```

L'appel à `MyIteratorMethod` n'exécute pas le corps de la méthode. À la place, l'appel retourne `IEnumerable<string>` dans la variable `elements`.

Dans une itération de la boucle `foreach`, la méthode MoveNext est appelée pour `elements`. Cet appel exécute le corps de `MyIteratorMethod` jusqu'à ce que l'instruction `yield return` suivante soit atteinte. L'expression retournée par l'instruction `yield return` détermine non seulement la valeur de la variable `element` pour la consommation par le corps de boucle, mais également la propriété Current de `elements`, qui est `IEnumerable<string>`.

À chaque itération suivante de la boucle `foreach`, l'exécution du corps de l'itérateur reprend à partir de l'emplacement où elle s'est interrompue, et s'arrête encore lorsqu'elle atteint une instruction `yield return`. La boucle `foreach` se termine lorsque à la fin de la méthode Iterator ou lorsqu'une instruction `yield break` est atteinte.

Exemple

L'exemple suivant comprend une instruction `yield return` située dans une boucle `for`. Chaque itération du corps d'instruction `foreach` dans la méthode `Main` crée un appel à la fonction d'itérateur `Power`. Chaque appel à la fonction d'itérateur continue vers l'exécution suivante de l'instruction `yield return`, qui se produit pendant l'itération suivante de la boucle `for`.

Le type de retour de la méthode Iterator est IEnumerable, qui est un type interface itérateur. Lorsque la méthode Iterator est appelée, elle retourne un objet énumérable contenant les puissances d'un nombre.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

Exemple

L'exemple suivant illustre un accesseur `get` qui est un itérateur. Dans cet exemple, chaque instruction `yield return` retourne une instance d'une classe définie par l'utilisateur.

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- foreach, in
- Iterators

Mots clés de requête (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette section contient les mots clés contextuels utilisés dans les expressions de requête.

Contenu de cette section

CLAUSE	DESCRIPTION
from	Spécifie une source de données et une variable de portée (semblable à une variable d'itération).
where	Filtre des éléments sources basés sur une ou plusieurs expressions booléennes séparées par des opérateurs AND et OR logiques (`&&` ou `||`).
select	Spécifie le type et la forme que les éléments de la séquence retournée auront une fois la requête exécutée.
groupe	Regroupe les résultats de la requête d'après une valeur de clé spécifiée.
into	Fournit un identificateur qui peut servir comme référence pour les résultats d'une clause join, group ou select.
TriPar	Trie des résultats de la requête par ordre croissant ou décroissant selon le comparateur par défaut du type d'élément.
join	Joint deux sources de données selon une comparaison d'égalité entre deux critères de comparaison spécifiés.
Explor	Introduit une variable de portée pour stocker des résultats de sous-expression dans une expression de requête.
in	Mot clé contextuel dans une clause join.
actif	Mot clé contextuel dans une clause join.
equals	Mot clé contextuel dans une clause join.
by	Mot clé contextuel dans une clause group.
ascending	Mot clé contextuel dans une clause orderby.
descending	Mot clé contextuel dans une clause orderby.

Voir aussi

- Mots clés C#

- LINQ (Language Integrated Query)
- LINQ en C#

from, clause (Référence C#)

16/12/2020 • 8 minutes to read • Edit Online

Une expression de requête doit commencer par une clause `from`. Une expression de requête peut également contenir des sous-requêtes qui commencent elles aussi par une clause `from`. La clause `from` spécifie les éléments suivants :

- La source de données sur laquelle la requête ou la sous-requête est exécutée.
- Une *variable de portée locale* qui représente chaque élément dans la séquence source.

La variable de portée et la source de données sont toutes les deux fortement typées. La source de données référencée dans la clause `from` doit avoir un type IEnumerable, IEnumerable<T> ou un type dérivé comme IQueryable<T>.

Dans l'exemple suivant, `numbers` est la source de données tandis que `num` est la variable de portée. Notez que les deux variables sont fortement typées même si le mot clé var est utilisé.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

Variable de portée

Le compilateur déduit le type de la variable de portée quand la source de données implémente IEnumerable<T>. Par exemple, si la source a un type `IEnumerable<Customer>`, la variable de portée est déduite comme étant `Customer`. Vous devez spécifier le type de façon explicite uniquement quand la source est un type `IEnumerable` non générique comme ArrayList. Pour plus d'informations, consultez Comment interroger une ArrayList avec LINQ.

Dans l'exemple précédent, `num` est déduit comme étant de type `int`. Comme la variable de portée est fortement typée, vous pouvez appeler des méthodes sur celle-ci ou l'utiliser dans d'autres opérations. Par exemple, au lieu d'écrire `select num`, vous pouvez écrire `select num.ToString()` pour que l'expression de requête retourne une séquence de chaînes à la place d'entiers. Vous pouvez également écrire `select num + 10` pour que l'expression retourne la séquence 14, 11, 13, 12, 10. Pour plus d'informations, consultez Select, clause.

La variable de portée est semblable à une variable d'itération dans une instruction foreach à une différence près, très importante : une variable de portée ne stocke jamais réellement de données de la source. Il s'agit juste d'une syntaxe pratique qui permet à la requête de décrire ce qui se produira lors de son exécution. Pour plus d'informations, consultez Introduction aux requêtes LINQ (C#).

Clauses from composées

Dans certains cas, chaque élément dans la séquence source peut être une séquence ou contenir une séquence. Par exemple, votre source de données peut être un `IEnumerable<Student>` où chaque objet Student dans la séquence contient une liste de notes d'examens. Pour accéder à la liste interne dans chaque élément `Student`, vous pouvez utiliser des clauses `from` composées. Cette technique est proche de celle consistant à utiliser des instructions foreach imbriquées. Vous pouvez ajouter des clauses where ou orderby à l'une et l'autre des clauses `from` pour filtrer les résultats. L'exemple suivant présente une séquence d'objets `Student` et chacun contient un `List` interne des entiers qui représentent des notes d'examens. Pour accéder à la liste interne, utilisez une clause `from` composée. Vous pouvez insérer des clauses entre les deux clauses `from` si nécessaire.

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
           new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
           new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

Utilisation de plusieurs clauses from pour effectuer des jointures

Une clause `from` composée est utilisée pour accéder aux collections internes dans une source de données unique. Toutefois, une requête peut également contenir plusieurs clauses `from` qui génèrent des requêtes supplémentaires de sources de données indépendantes. Cette technique vous permet d'effectuer certains types d'opérations de jointure qui ne sont pas possibles en utilisant la clause join.

L'exemple suivant montre comment utiliser deux clauses `from` pour former une jointure croisée complète de

deux sources de données.

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

Pour plus d'informations sur les opérations de jointure qui utilisent plusieurs clauses `from`, consultez Effectuer des jointures externes gauches.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ (Language-Integrated Query)

where, clause (Référence C#)

16/12/2020 • 4 minutes to read • Edit Online

La clause `where` est utilisée dans une expression de requête pour spécifier les éléments de la source de données qui seront retournés dans l'expression de requête. Elle applique une condition booléenne (un *prédicat*) à chaque élément source (référencé par la variable de portée) et retourne ceux pour lesquels la condition spécifiée est remplie. Une expression de requête unique peut contenir plusieurs clauses `where` et une clause unique peut contenir plusieurs sous-expressions de prédicat.

Exemple

Dans l'exemple suivant, la clause `where` élimine par filtrage tous les nombres excepté ceux inférieurs à cinq. Si vous supprimez la clause `where`, tous les nombres de la source de données seront retournés. L'expression `num < 5` est le prédicat qui est appliqué à chaque élément.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Exemple

Dans une seule `where` clause, vous pouvez spécifier autant de prédicats que nécessaire à l'aide des `&&` opérateurs et `||` . Dans l'exemple suivant, la requête spécifie deux prédicats pour sélectionner uniquement les nombres pairs inférieurs à cinq.

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

Exemple

Une clause `where` peut contenir une ou plusieurs méthodes qui retournent des valeurs booléennes. Dans l'exemple suivant, la clause `where` utilise une méthode pour déterminer si la valeur actuelle de la variable de portée est paire ou impaire.

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

Notes

La clause `where` est un mécanisme de filtrage. Elle peut être placée à presque n'importe quel endroit d'une expression de requête, sauf qu'elle ne peut pas être la première ni la dernière clause. Une clause `where` peut apparaître avant ou après une clause group selon que vous devez filtrer les éléments sources avant ou après leur regroupement.

Si un prédicat spécifié n'est pas valide pour les éléments de la source de données, une erreur de compilation est générée. C'est l'un des avantages du contrôle de type fort fourni par LINQ.

Au moment de la compilation, le mot clé `where` est converti en appel à la méthode d'opérateur de requête standard Where.

Voir aussi

- Mots clés de requête (LINQ)
- from, clause
- clause SELECT
- Filtrage des données
- LINQ en C#
- LINQ (Language-Integrated Query)

select, clause (Référence C#)

16/12/2020 • 7 minutes to read • Edit Online

Dans une expression de requête, la clause `select` spécifie le type de valeurs qui sont générées quand la requête est exécutée. Le résultat est basé sur l'évaluation de toutes les clauses précédentes et sur toutes les expressions de la clause `select` elle-même. Une expression de requête doit se terminer par une clause `select` ou par une clause group.

L'exemple suivant présente une clause `select` simple dans une expression de requête.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

Le type de la séquence générée par la clause `select` détermine le type de la variable de requête `queryHighScores`. Dans le cas le plus simple, la clause `select` spécifie uniquement la variable de portée. La séquence retournée contient alors des éléments du même type que la source de données. Pour plus d'informations, consultez relations de types dans les opérations de requête LINQ. Toutefois, la clause `select` fournit également un mécanisme puissant pour transformer (ou *projeter*) les données sources en nouveaux types. Pour plus d'informations, consultez Transformations de données avec LINQ (C#).

Exemple

L'exemple suivant affiche l'ensemble des différentes formes que peut prendre une clause `select`. Dans chaque requête, notez la relation entre la `select` clause et le type de la *variable de requête* (`studentQuery1` , `studentQuery2` , etc.).

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
            public int ID { get; set; }
            public List<int> Scores;
            public ContactInfo GetContactInfo(SelectSample2 app, int id)
            {
```

```
            {
                ContactInfo cInfo =
                    (from ci in app.contactList
                    where ci.ID == id
                    select ci)
                    .FirstOrDefault();

                return cInfo;
            }

            public override string ToString()
            {
                return First + " " + Last + ":" + ID;
            }
        }

        public class ContactInfo
        {
            public int ID { get; set; }
            public string Email { get; set; }
            public string Phone { get; set; }
            public override string ToString() { return Email + "," + Phone; }
        }

        public class ScoreInfo
        {
            public double Average { get; set; }
            public int ID { get; set; }
        }

        // The primary data source
        List<Student> students = new List<Student>()
        {
             new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92,
81, 60}},
             new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39}},
             new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65,
91}},
             new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85,
82}},
        };

        // Separate data source for contact info.
        List<ContactInfo> contactList = new List<ContactInfo>()
        {
            new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
            new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
            new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
            new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
        };

        static void Main(string[] args)
        {
            SelectSample2 app = new SelectSample2();

            // Produce a filtered sequence of unmodified Students.
            IEnumerable<Student> studentQuery1 =
                from student in app.students
                where student.ID > 111
                select student;

            Console.WriteLine("Query1: select range_variable");
            foreach (Student s in studentQuery1)
            {
                Console.WriteLine(s.ToString());
            }

            // Produce a filtered sequence of elements that contain
            // only one property of each Student
```

```
// only one property of each Student.
IEnumerable<String> studentQuery2 =
    from student in app.students
    where student.ID > 111
    select student.Last;

Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
foreach (string s in studentQuery2)
{
    Console.WriteLine(s);
}

// Produce a filtered sequence of objects created by
// a method call on each Student.
IEnumerable<ContactInfo> studentQuery3 =
    from student in app.students
    where student.ID > 111
    select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
foreach (ContactInfo ci in studentQuery3)
{
    Console.WriteLine(ci.ToString());
}

// Produce a filtered sequence of ints from
// the internal array inside each Student.
IEnumerable<int> studentQuery4 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
foreach (int i in studentQuery4)
{
    Console.WriteLine("First score = {0}", i);
}

// Produce a filtered sequence of doubles
// that are the result of an expression.
IEnumerable<double> studentQuery5 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0] * 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");
foreach (double d in studentQuery5)
{
    Console.WriteLine("Adjusted first score = {0}", d);
}

// Produce a filtered sequence of doubles that are
// the result of a method call.
IEnumerable<double> studentQuery6 =
    from student in app.students
    where student.ID > 111
    select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");
foreach (double d in studentQuery6)
{
    Console.WriteLine("Average = {0}", d);
}

// Produce a filtered sequence of anonymous types
// that contain only two properties from each Student.
var studentQuery7 =
    from student in app.students
    where student.ID > 111
```

```
            select new { student.First, student.Last };

        Console.WriteLine("\r\n studentQuery7: select new anonymous type");
        foreach (var item in studentQuery7)
        {
            Console.WriteLine("{0}, {1}", item.Last, item.First);
        }

        // Produce a filtered sequence of named objects that contain
        // a method return value and a property from each Student.
        // Use named types if you need to pass the query variable
        // across a method boundary.
        IEnumerable<ScoreInfo> studentQuery8 =
            from student in app.students
            where student.ID > 111
            select new ScoreInfo
            {
                Average = student.Scores.Average(),
                ID = student.ID
            };

        Console.WriteLine("\r\n studentQuery8: select new named type");
        foreach (ScoreInfo si in studentQuery8)
        {
            Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
        }

        // Produce a filtered sequence of students who appear on a contact list
        // and whose average is greater than 85.
        IEnumerable<ContactInfo> studentQuery9 =
            from student in app.students
            where student.Scores.Average() > 85
            join ci in app.contactList on student.ID equals ci.ID
            select ci;

        Console.WriteLine("\r\n studentQuery9: select result of join clause");
        foreach (ContactInfo ci in studentQuery9)
        {
            Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
        }
    }
/* Output
    Query1: select range_variable
    Claire O'Donnell:112
    Sven Mortensen:113
    Cesar Garcia:114

    studentQuery2: select range_variable.Property
    O'Donnell
    Mortensen
    Garcia

    studentQuery3: select range_variable.Method
    ClaireO@Contoso.com,206-555-0298
    SvenMort@Contoso.com,206-555-1130
    CesarGar@Contoso.com,206-555-0521

    studentQuery4: select range_variable[index]
    First score = 75
    First score = 88
    First score = 97

    studentQuery5: select expression
    Adjusted first score = 82.5
```

```
        Adjusted first score = 96.8
        Adjusted first score = 106.7

        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

Comme indiqué dans `studentQuery8` dans l'exemple précédent, vous pouvez parfois souhaiter que les éléments de la séquence retournée contiennent uniquement un sous-ensemble des propriétés des éléments sources. En limitant au maximum la séquence retournée, vous pouvez réduire les besoins en ressources mémoire et augmenter la vitesse d'exécution de la requête. Pour ce faire, créez un type anonyme dans la clause `select` et utilisez un initialiseur d'objet pour l'initialiser avec les propriétés appropriées de l'élément source. Pour obtenir un exemple de la procédure à suivre, consultez Initialiseurs d'objet et de collection.

Notes

Au moment de la compilation, la clause `select` traduite en un appel de méthode à l'opérateur de requête standard Select.

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- from, clause
- partial, méthode (Référence C#)
- Types anonymes
- LINQ en C#
- LINQ (Language-Integrated Query)

group, clause (Référence C#)

16/12/2020 • 12 minutes to read • Edit Online

La clause `group` retourne une séquence d'objets IGrouping<TKey,TElement> qui contient zéro, un ou plusieurs éléments qui correspondent à la valeur de clé du groupe. Par exemple, vous pouvez regrouper une séquence de chaînes en fonction de la première lettre de chaque chaîne. Dans ce cas, la première lettre est la clé, elle a le type char et elle est stockée dans la propriété `Key` de chaque objet IGrouping<TKey,TElement>. Le compilateur déduit le type de la clé.

Vous pouvez terminer une expression de requête avec une clause `group`, comme illustré dans l'exemple suivant :

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Si vous souhaitez effectuer des opérations de requête supplémentaires sur chaque groupe, vous pouvez spécifier un identificateur temporaire en utilisant le mot clé contextuel into. Quand vous utilisez `into`, vous devez continuer la requête et finalement la terminer avec une instruction `select` ou une autre clause `group`, comme illustré dans l'extrait suivant :

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

Des exemples d'utilisation plus complets de `group` avec et sans `into` sont fournis dans la section Exemple de cet article.

Énumération des résultats d'une requête de groupe

Étant donné que les objets IGrouping<TKey,TElement> générés par une requête `group` sont essentiellement une liste de listes, vous devez utiliser une boucle foreach imbriquée pour accéder aux éléments dans chaque groupe. La boucle externe itère les clés de groupe, et la boucle interne itère chaque élément dans le groupe proprement dit. Un groupe peut avoir une clé mais pas d'éléments. Voici la boucle `foreach` qui exécute la requête dans les exemples de code précédents :

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Types de clés

Les clés de groupes peuvent être de tout type, comme une chaîne, un type numérique intégré, ou un type nommé ou un type anonyme défini par l'utilisateur.

Regroupement par chaîne

Les exemples de code précédents utilisaient un `char`. On aurait facilement pu spécifier une clé de chaîne à la place, par exemple le nom de famille complet :

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Regroupement par valeur booléenne

L'exemple suivant illustre l'utilisation d'une valeur booléenne pour une clé afin de répartir les résultats en deux groupes. Notez que la valeur est générée par une sous-expression dans la clause `group`.

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91,
39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91,
95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First,
student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Regroupement par plage numérique

L'exemple suivant utilise une expression pour créer des clés de groupes numériques qui représentent une plage de centiles. Notez l'utilisation de let comme emplacement pratique pour stocker un résultat d'appel de méthode, qui vous évite d'avoir à appeler deux fois la méthode dans la clause `group`. Pour plus d'informations sur la façon d'utiliser des méthodes en toute sécurité dans les expressions de requête, consultez gérer les exceptions dans les expressions de requête.

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91,
39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91,
95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First,
student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
```

```
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
     Students with an average between 80 and 90
       Garcia, Debra:88.25
     Students with an average between 90 and 100
       Mortensen, Sven:93.5
 */
```

Regroupement par clés composites

Utilisez une clé composite quand vous souhaitez regrouper des éléments en fonction de plusieurs clés. Vous créez une clé composite en utilisant un type anonyme ou un type nommé pour contenir l'élément clé. Dans l'exemple suivant, supposons qu'une classe `Person` a été déclarée avec les membres nommés `surname` et `city`. La clause `group` provoque la création d'un groupe distinct pour chaque ensemble de personnes ayant le même nom de famille et la même ville.

```
group person by new {name = person.surname, city = person.city};
```

Utilisez un type nommé si vous devez passer la variable de requête à une autre méthode. Créez une classe spéciale à l'aide de propriétés implémentées automatiquement pour les clés, puis substituez les méthodes Equals et GetHashCode. Vous pouvez également utiliser un struct, auquel cas vous n'êtes pas obligé de substituer ces méthodes. Pour plus d'informations, consultez comment implémenter une classe Lightweight avec des propriétés implémentées automatiquement et Comment interroger des fichiers dupliqués dans une arborescence de répertoires. Ce dernier article contient un exemple de code qui montre comment utiliser une clé composite avec un type nommé.

Exemple

L'exemple suivant montre le modèle standard pour organiser des données sources en groupes quand aucune logique de requête supplémentaire n'est appliquée aux groupes. Il s'agit alors d'un regroupement sans continuation. Les éléments du tableau de chaînes sont regroupés en fonction de leur première lettre. Le résultat de la requête est un type IGrouping<TKey,TElement> contenant une propriété `Key` publique de type `char` et une collection IEnumerable<T> qui contient chaque élément du regroupement.

Le résultat d'une clause `group` est une séquence de séquences. Ainsi, pour accéder aux différents éléments de chaque groupe retourné, vous devez utiliser une boucle `foreach` imbriquée à l'intérieur de la boucle qui itère les clés de groupe, comme illustré dans l'exemple suivant.

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Exemple

Cet exemple montre comment exécuter une logique supplémentaire sur les groupes après les avoir créés, à l'aide une *continuation* avec `into`. Pour plus d'informations, consultez into. L'exemple suivant interroge chaque groupe pour sélectionner uniquement ceux dont la valeur de clé est une voyelle.

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese",
"elephant", "umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Notes

Lors de la compilation, les clauses `group` sont traduites en appels à la méthode GroupBy.

Voir aussi

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Mots clés de requête
- LINQ (Language-Integrated Query)
- Créer un groupe imbriqué
- Regrouper les résultats d'une requête
- Effectuer une sous-requête sur une opération de regroupement

into (Référence C#)

16/12/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `into` permet de créer un identificateur temporaire pour stocker les résultats d'une clause group, join ou select dans un nouvel identificateur. Cet identificateur peut lui-même être un générateur pour d'autres commandes de requête. Quand il est utilisé dans une clause `group` ou `select`, le nouvel identificateur est parfois appelé *continuation*.

Exemple

L'exemple suivant illustre l'utilisation du mot clé `into` pour activer un identificateur temporaire `fruitGroup` dont le type déduit est `IGrouping`. En utilisant l'identificateur, vous pouvez appeler la méthode Count dans chaque groupe et sélectionner uniquement les groupes qui contiennent deux mots ou plus.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

L'utilisation de `into` dans une clause `group` n'est nécessaire que si vous voulez effectuer des opérations de requête supplémentaires dans chaque groupe. Pour plus d'informations, consultez Group, clause.

Pour obtenir un exemple d'utilisation de `into` dans une clause `join`, consultez join, clause.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ en C#
- group, clause

orderby, clause (Référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Dans une expression de requête, la clause `orderby` a pour effet de trier la séquence ou la sous-séquence (groupe) retournée par ordre croissant ou décroissant. Il est possible de spécifier plusieurs clés de façon à effectuer une ou plusieurs opérations de tri secondaire. Le tri est effectué par le comparateur par défaut du type de l'élément. L'ordre de tri par défaut est croissant. Vous pouvez aussi spécifier un comparateur personnalisé. Cependant, sa disponibilité est soumise à l'utilisation d'une syntaxe fondée sur une méthode. Pour plus d'informations, consultez Tri des données.

Exemple

Dans l'exemple suivant, la première requête trie les mots par ordre alphabétique à partir de A, tandis que la deuxième requête trie les mêmes mots par ordre décroissant. (Le mot clé `ascending` est la valeur de tri par défaut et peut être omis.)

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Exemple

L'exemple suivant effectue un tri principal sur les noms des étudiants, puis un tri secondaire sur leurs prénoms.

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;
    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the names
        // of all students so that they will be in alphabetical order after they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last, student.First);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
```

```
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

Remarques

Au moment de la compilation, la clause `orderby` est traduite en appel à la méthode OrderBy. Les différentes clés présentes dans la clause `orderby` sont traduites en appels à la méthode ThenBy.

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- LINQ en C#
- group, clause
- LINQ (Language-Integrated Query)

join, clause (référence C#)

02/11/2020 • 14 minutes to read • Edit Online

La clause `join` est utile pour associer des éléments de différentes séquences sources qui n'ont pas de relation directe dans le modèle objet. Le seul impératif est que les éléments de chaque source partagent une valeur dont l'égalité peut être comparée. Par exemple, un distributeur de produits alimentaires peut avoir une liste de fournisseurs d'un certain produit et une liste d'acheteurs. Une clause `join` peut être utilisée par exemple pour créer une liste des fournisseurs et des acheteurs de ce produit qui se trouvent tous dans la même région spécifiée.

Une clause `join` prend deux séquences sources comme entrée. Les éléments de chaque séquence doivent être ou contenir une propriété qui peut être comparée à une propriété correspondante dans l'autre séquence. La clause `join` utilise le mot clé spécial `equals` pour comparer les clés spécifiées et déterminer si elles sont égales. Toutes les jointures effectuées par la clause `join` sont des équijointures. La forme de la sortie d'une clause `join` dépend du type spécifique de jointure que vous effectuez. Les trois types de jointure les plus courants sont les suivants :

- Jointure interne
- Jointure groupée
- Jointure externe gauche

Jointure interne

L'exemple suivant montre une équijointure interne simple. Cette requête produit une séquence plate de paires « nom de produit / catégorie ». La même chaîne de catégorie apparaît dans plusieurs éléments. Si un élément de `categories` n'a pas de `products` correspondant, cette catégorie n'apparaît pas dans les résultats.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

Pour plus d'informations, consultez Effectuer des jointures internes.

Jointure groupée

Une clause `join` avec une expression `into` est appelée une jointure groupée.

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

Une jointure groupée produit une séquence de résultats hiérarchique, qui associe des éléments de la séquence source de gauche à un ou plusieurs éléments correspondants de la séquence source de droite. Une jointure groupée n'a pas d'équivalent en termes relationnels ; il s'agit en fait d'une séquence de tableaux d'objets.

Si aucun élément de la séquence source de droite n'est trouvé en correspondance avec un élément de la source de gauche, la clause `join` produit un tableau vide pour cet élément. Par conséquent, la jointure groupée est fondamentalement une équijointure interne, excepté que la séquence de résultats est organisée en groupes.

Si vous sélectionnez les résultats d'une jointure groupée, vous pouvez accéder aux éléments, mais vous ne pouvez pas identifier leur clé de correspondance. Par conséquent, il est généralement plus utile de sélectionner les résultats de la jointure groupée dans un nouveau type qui a aussi le nom de clé, comme illustré dans l'exemple précédent.

Bien sûr, vous pouvez aussi utiliser le résultat d'une jointure groupée comme générateur d'une autre sous-requête :

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

Pour plus d'informations, consultez Effectuer des jointures groupées.

Jointure externe gauche

Dans une jointure externe gauche, tous les éléments de la séquence source de gauche sont retournés, même si aucun élément correspondant ne se trouve dans la séquence de droite. Pour effectuer une jointure externe gauche dans LINQ, utilisez la `DefaultIfEmpty` méthode en association avec une jointure groupée pour spécifier un élément côté droit par défaut à produire si un élément côté gauche n'a pas de correspondance. Vous pouvez utiliser `null` comme valeur par défaut pour tous les types référence ou vous pouvez spécifier un type par défaut défini par l'utilisateur. L'exemple suivant montre un type par défaut défini par l'utilisateur :

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

Pour plus d'informations, consultez Effectuer des jointures externes gauches.

Opérateur Égal

Une clause `join` effectue une équijointure. En d'autres termes, vous pouvez baser les correspondances seulement sur l'égalité de deux clés. Les autres types de comparaisons, comme « supérieur à » ou « différent de », ne sont pas pris en charge. Pour indiquer clairement que toutes les jointures sont des équijointures, la clause `join` utilise le mot clé `equals` au lieu de l'opérateur `==`. Le mot clé `equals` peut être utilisé seulement dans une clause `join` et diffère de l'opérateur `==` sur un point important. Avec `equals`, la clé gauche consomme la séquence source externe et la clé droite consomme la source interne. La source externe est seulement dans l'étendue du côté gauche de `equals` et la séquence source interne est seulement dans l'étendue du côté droit.

Non-équijointures

Vous pouvez effectuer des non-équijointures, des jointures croisées et d'autres opérations de jointure

personnalisées en utilisant plusieurs clauses `from` pour introduire indépendamment de nouvelles séquences dans une requête. Pour plus d'informations, consultez Effectuer des opérations de jointure personnalisées.

Jointures sur des collections d'objets et sur des tables relationnelles

Dans une expression de requête LINQ, les opérations de jointure sont effectuées sur les collections d'objets. Les collections d'objets ne peuvent pas être « jointes » exactement de la même façon que deux tables relationnelles. Dans LINQ, les `join` clauses explicites sont uniquement requises lorsque deux séquences sources ne sont pas liées par une relation. Quand vous travaillez avec LINQ to SQL, les tables avec des clés étrangères sont représentées dans le modèle objet en tant que propriétés de la table principale. Par exemple, dans la base de données Northwind, la table Customer a une relation de clé étrangère avec la table Orders. Quand vous mappez les tables au modèle objet, la classe Customer a une propriété Orders qui contient la collection de commandes associées à ce client. En réalité, la jointure a déjà été effectuée pour vous.

Pour plus d'informations sur l'interrogation de tables liées dans le contexte de LINQ to SQL, consultez Procédure : mapper des relations de base de données.

Clés composites

Vous pouvez tester l'égalité de plusieurs valeurs en utilisant une clé composite. Pour plus d'informations, consultez Effectuer des opérations de jointure à l'aide de clés composites. Vous pouvez aussi utiliser des clés composites dans une clause `group`.

Exemple

L'exemple suivant compare les résultats d'une jointure interne, d'une jointure groupée et d'une jointure externe gauche sur les mêmes sources de données en utilisant les mêmes clés de correspondance. Du code supplémentaire est ajouté à ces exemples pour clarifier les résultats dans l'affichage de la console.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
```

```
        List<Product> products = new List<Product>()
       {
          new Product {Name="Cola",  CategoryID=001},
          new Product {Name="Tea",  CategoryID=001},
          new Product {Name="Mustard", CategoryID=002},
          new Product {Name="Pickles", CategoryID=002},
          new Product {Name="Carrots", CategoryID=003},
          new Product {Name="Bok Choy", CategoryID=003},
          new Product {Name="Peaches", CategoryID=005},
          new Product {Name="Melons", CategoryID=005},
        };
        #endregion

        static void Main(string[] args)
        {
            JoinDemonstration app = new JoinDemonstration();

            app.InnerJoin();
            app.GroupJoin();
            app.GroupInnerJoin();
            app.GroupJoin3();
            app.LeftOuterJoin();
            app.LeftOuterJoin2();

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }

        void InnerJoin()
        {
            // Create the query that selects
            // a property from each element.
            var innerJoinQuery =
               from category in categories
               join prod in products on category.ID equals prod.CategoryID
               select new { Category = category.ID, Product = prod.Name };

            Console.WriteLine("InnerJoin:");
            // Execute the query. Access results
            // with a simple foreach statement.
            foreach (var item in innerJoinQuery)
            {
                Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
            }
            Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
            Console.WriteLine(System.Environment.NewLine);
        }

        void GroupJoin()
        {
            // This is a demonstration query to show the output
            // of a "raw" group join. A more typical group join
            // is shown in the GroupInnerJoin method.
            var groupJoinQuery =
               from category in categories
               join prod in products on category.ID equals prod.CategoryID into prodGroup
               select prodGroup;

            // Store the count of total items (for demonstration only).
            int totalItems = 0;

            Console.WriteLine("Simple GroupJoin:");

            // A nested foreach statement is required to access group items.
            foreach (var prodGrouping in groupJoinQuery)
            {
                Console.WriteLine("Group:");
                foreach (var item in prodGrouping)
```

```
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName, item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
```

```
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID =
category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems,
leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.


Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
```

```
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
```

```
Peaches    5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Remarques

Une clause `join` qui n'est pas suivie de `into` se traduit par l'appel de la méthode Join. Une clause `join` qui n'est pas suivie de `into` se traduit par l'appel de la méthode GroupJoin.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ (Language-Integrated Query)
- Opérations de jointure
- group, clause
- Effectuer des jointures externes gauches
- Effectuer des jointures internes
- Effectuer des jointures groupées
- Classer les résultats d'une clause join
- Effectuer des jointures à l'aide de clés composites
- Systèmes de base de données compatibles pour Visual Studio

let, clause (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Dans une expression de requête, il est parfois utile de stocker le résultat d'une sous-expression pour pouvoir l'utiliser dans des clauses ultérieures. Pour cela, vous pouvez utiliser le mot clé `let`, qui crée une variable de portée et l'initialise avec le résultat de l'expression que vous fournissez. Une fois initialisée avec une valeur, la variable de portée ne peut pas être utilisée pour stocker une autre valeur. Cependant, si la variable de portée contient un type requêtable, elle peut être interrogée.

Exemple

Dans l'exemple suivant, la clause `let` est utilisée de deux façons différentes :

1. Pour créer un type énumérable qui peut lui-même être interrogé.
2. Pour permettre à la requête d'appeler `ToLower` une seule fois sur la variable de portée `word`. Si vous n'utilisez pas `let`, vous devez appeler `ToLower` dans chaque prédicat de la clause `where`.

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- LINQ en C#
- LINQ (Language-Integrated Query)
- Gérer des exceptions dans des expressions de requête

ascending (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `ascending` est utilisé dans la clause orderby dans les expressions de requête afin de spécifier que l'ordre de tri est du plus petit au plus grand. `ascending` étant l'ordre de tri par défaut, vous n'avez pas à le spécifier.

Exemple

L'exemple suivant illustre l'utilisation de `ascending` dans une clause orderby.

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

Voir aussi

- Référence C#
- LINQ en C#
- descending

descending (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `descending` est utilisé dans la clause orderby dans les expressions de requête afin de spécifier que l'ordre de tri est du plus grand au plus petit.

Exemple

L'exemple suivant illustre l'utilisation de `descending` dans une clause orderby.

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

Voir aussi

- Référence C#
- LINQ en C#
- ascending

on (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `on` est utilisée dans la clause join d'une expression de requête pour spécifier la condition de jointure.

Exemple

L'exemple suivant illustre l'utilisation de `on` dans une clause `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Voir aussi

- Référence C#
- LINQ (Language-Integrated Query)

equals (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `equals` est utilisé dans une clause `join` dans une expression de requête pour comparer les éléments de deux séquences. Pour plus d'informations, consultez join, clause.

Exemple

L'exemple suivant illustre l'utilisation du mot clé `equals` dans une clause `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Voir aussi

- LINQ (Language-Integrated Query)

by (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le mot clé contextuel `by` est utilisé dans la clause `group` dans une expression de requête pour spécifier la façon dont les éléments retournés doivent être groupés. Pour plus d'informations, consultez Group, clause.

Exemple

L'exemple suivant illustre l'utilisation du mot clé contextuel `by` dans une clause `group` pour spécifier que les étudiants doivent être groupés en fonction de la première lettre de leur nom.

```
var query = from student in students
            group student by student.LastName[0];
```

Voir aussi

- LINQ en C#

in (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Le `in` mot clé est utilisé dans les contextes suivants :

- paramètre de type générique dans les interfaces et les délégués génériques.
- Comme modificateur de paramètre, qui vous permet de passer un argument à une méthode par référence plutôt que par valeur.
- Instructions foreach.
- Clauses from dans les expressions de requête LINQ.
- Clauses join dans les expressions de requête LINQ.

Voir aussi

- Mots clés C#
- Référence C#

Opérateurs et expressions c# (référence C#)

16/12/2020 • 9 minutes to read • Edit Online

C# fournit un certain nombre d'opérateurs. Un grand nombre d'entre eux sont pris en charge par les types intégrés et vous permettent d'effectuer des opérations de base avec des valeurs de ces types. Ces opérateurs incluent les groupes suivants :

- Opérateurs arithmétiques qui effectuent des opérations arithmétiques avec des opérandes numériques
- Opérateurs de comparaison qui comparent des opérandes numériques
- Opérateurs logiques booléens qui effectuent des opérations logiques avec des `bool` opérandes
- Opérateurs de bits et de décalage qui effectuent des opérations de bits ou de décalage avec les opérandes des types intégraux
- Opérateurs d'égalité qui vérifient si leurs opérandes sont égaux ou non

En général, vous pouvez surcharger ces opérateurs, autrement dit, spécifier le comportement de l'opérateur pour les opérandes d'un type défini par l'utilisateur.

Les expressions C# les plus simples sont des littéraux (par exemple, des nombres entiers et réels) et des noms de variables. Vous pouvez les combiner dans des expressions complexes à l'aide d'opérateurs. La priorité et l' associativité des opérateurs déterminent l'ordre dans lequel les opérations dans une expression sont exécutées. Vous pouvez utiliser des parenthèses pour changer l'ordre d'évaluation imposé par la priorité et l'associativité de l'opérateur.

Dans le code suivant, des exemples d'expressions se trouvent dans la partie droite des affectations :

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

En règle générale, une expression produit un résultat et peut être incluse dans une autre expression. Un `void` appel de méthode est un exemple d'expression qui ne produit pas de résultat. Il ne peut être utilisé qu'en tant qu' instruction, comme le montre l'exemple suivant :

```
Console.WriteLine("Hello, world!");
```

Voici d'autres genres d'expressions fournis par C# :

- Expressions de chaîne interpolées qui fournissent une syntaxe pratique pour créer des chaînes mises en forme :

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- Expressions lambda qui vous permettent de créer des fonctions anonymes :

```
int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- Expressions de requête qui vous permettent d'utiliser directement des fonctionnalités de requête en C# :

```
var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

Vous pouvez utiliser une définition de corps d'expression pour fournir une définition concise d'une méthode, d'un constructeur, d'une propriété, d'un indexeur ou d'un finaliseur.

Précédence des opérateurs

Dans une expression avec plusieurs opérateurs, les opérateurs avec une priorité plus élevée sont évalués avant les opérateurs avec une priorité moins élevée. Dans l'exemple suivant, la multiplication est effectuée en premier, car elle a une priorité plus élevée que l'addition :

```
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Utilisez des parenthèses pour changer l'ordre d'évaluation imposé par la précédence des opérateurs :

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

La table suivante répertorie les opérateurs C# de la priorité la plus élevée à la plus basse. Les opérateurs dans chaque ligne sont de priorité égale.

OPÉRATEURS	CATÉGORIE OU NOM
x. y, f (x), a[i], `x?.y` , `x?[y]` , x + +, x--, x !, New, typeof, coché, désactivé, default, nameof, Delegate, sizeof, stackalloc, x->y	Principal

OPÉRATEURS	CATÉGORIE OU NOM
+ x, -x, ! x, ~ x, + + x, --x, ^ x, (T) x, await, &x, * x, true et false	Unaire
x.. y	Plage
switch	Expression `switch`
with	Expression `with`
x * y, x / y, x % y	Multiplicatif
x + y, x – y	Additive
x << y, x >> y	Maj
x < y, x > y, x <= y, x > = y, est, comme	Relations et test de type
x = = y, x ! = y	Égalité
`x & y`	Logique booléen AND ou logique bitwise AND
`x ^ y`	Logique booléen XOR ou logique bitwise XOR
`x | y`	Logique booléen OR ou logique bitwise OR
x && y	AND conditionnel
x || y	OR conditionnel
x ?? y	Opérateur de fusion de Null
c ? t : f	Opérateur conditionnel
x = y, x + = y, x-= y, x * = y, x/= y, x% = y, x &= y, x |= y, x ^ = y, x <<= y, x >>= y, x ?? = y, =>	Affectation et déclaration lambda

Associativité des opérateurs

Quand les opérateurs sont de même priorité, l'associativité des opérateurs détermine l'ordre dans lequel les opérations sont effectuées :

- Les opérateurs *associatifs sur leur gauche* sont évalués dans l'ordre, de gauche à droite. À l'exception des opérateurs d' assignation et des opérateurs de fusion Null, tous les opérateurs binaires sont associatifs à gauche. Par exemple, `a + b - c` est évalué comme étant `(a + b) - c`.
- Les opérateurs *associatifs sur leur droite* sont évalués dans l'ordre, de droite à gauche. Les opérateurs d'assignation, les opérateurs de fusion Null et l' opérateur `?:` conditionnel sont associatifs à droite. Par exemple, `x = y = z` est évalué comme étant `x = (y = z)`.

Utilisez des parenthèses pour changer l'ordre d'évaluation imposé par l'associativité des opérateurs :

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Évaluation des opérandes

Sans rapport avec la priorité et l'associativité des opérateurs, les opérandes dans une expression sont évalués de gauche à droite. Les exemples suivants illustrent l'ordre dans lequel les opérateurs et les opérandes sont évalués :

EXPRESSION	ORDRE D'ÉVALUATION
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

En règle générale, tous les opérandes d'opérateur sont évalués. Toutefois, certains opérateurs évaluent les opérandes de manière conditionnelle. Autrement dit, la valeur de l'opérande le plus à gauche d'un opérateur de ce type définit si les autres opérandes doivent être évalués. Ces opérateurs sont les opérateurs logiques and (`&&`) et or `||` () conditionnels, les opérateurs de fusion Null `??` et `??=` , les opérateurs conditionnels null `?.` et `?[]` , ainsi que l' opérateur `?:` conditionnel . Pour plus d'informations, consultez la description de chaque opérateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Expressions
- Opérateurs

Voir aussi

- Informations de référence sur C#
- Surcharge d'opérateur
- Arborescences d'expressions

Opérateurs arithmétiques (référence C#)

02/11/2020 • 17 minutes to read • Edit Online

Les opérateurs suivants effectuent des opérations arithmétiques avec des opérandes de types numériques :

- Opérateurs unaires `++` (incrémentation), `--` (décrémentation), `+` (plus)et `-` (moins)
- Opérateurs binaires `*` (multiplication), (`/` Division), `%` (reste), `+` (addition)et `-` (soustraction)

Ces opérateurs sont pris en charge par tous les types numériques intégraux et à virgule flottante .

Dans le cas des types intégraux, ces opérateurs (à l'exception des `++` `--` opérateurs et) sont définis pour les `int` `uint` types,, `long` et `ulong` . Lorsque les opérandes sont d'autres types intégraux (`sbyte` , `byte` ,, `short` `ushort` ou `char`), leurs valeurs sont converties vers le `int` type, qui est également le type de résultat d'une opération. Quand les opérandes sont de types intégraux ou à virgule flottante différents, leurs valeurs sont converties dans le type conteneur le plus proche, si un tel type existe. Pour plus d'informations, consultez la section Promotions numériques de la spécification du langage C#. Les `++` `--` opérateurs et sont définis pour tous les types numériques intégraux et à virgule flottante, ainsi que le type char .

Opérateur d'incrémentation ++

L'opérateur d'incrémentation unaire `++` incrémente son opérande de 1. L'opérande doit être une variable, un accès propriété ou un accès indexeur.

L'opérateur d'incrémentation est pris en charge sous deux formes : l'opérateur d'incrémentation suffixé, `x++`, et l'opérateur d'incrémentation préfixé, `++x`.

Opérateur d'incrément suffixé

Le résultat de `x++` est la valeur de `x` *avant* l'opération, comme le montre l'exemple suivant :

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Opérateur d'incrémentation préfixé

Le résultat de `++x` est la valeur de `x` *après* l'opération, comme le montre l'exemple suivant :

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Opérateur de décrémentation --

L'opérateur de décrémentation unaire `--` décrémente son opérande de 1. L'opérande doit être une variable, un accès propriété ou un accès indexeur.

L'opérateur de décrémentation est pris en charge sous deux formes : l'opérateur de décrémentation suffixé, `x--`, et l'opérateur de décrémentation préfixé, `--x`.

Opérateur de décrémentation suffixé

Le résultat de `x--` est la valeur de `x` *avant* l'opération, comme le montre l'exemple suivant :

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Opérateur de décrémentation préfixé

Le résultat de `--x` est la valeur de `x` *après* l'opération, comme le montre l'exemple suivant :

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Opérateurs plus et moins unaires

L'opérateur unaire `+` retourne la valeur de son opérande. L'opérateur unaire `-` calcule la négation numérique de son opérande.

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

Le type ULong ne prend pas en charge l'opérateur unaire `-` .

Opérateur de multiplication *

L'opérateur de multiplication `*` calcule le produit de ses opérandes :

```
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

L'opérateur unaire `*` est l'opérateur d'indirection de pointeur.

Opérateur de division /

L'opérateur de division `/` divise son opérande de partie gauche par son opérande de partie droite.

Division d'entier

Pour les opérandes des types entiers, le résultat de l'opérateur `/` est de type entier et égal au quotient de deux opérandes arrondis vers le zéro :

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

Pour obtenir le quotient de deux opérandes comme un nombre à virgule flottante, utilisez le type `float`, `double` ou `decimal` :

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

Division à virgule flottante

Pour les types `float`, `double` et `decimal`, le résultat de l'opérateur `/` est le quotient de deux opérandes :

```
Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

Si l'un des opérandes est `decimal`, un autre opérande ne peut être `float` ni `double`, car ni `float` ni `double` ne sont implicitement convertibles en `decimal`. Vous devez convertir explicitement l'opérande `float` ou `double` en type `decimal`. Pour plus d'informations sur les conversions entre les types numériques, consultez conversions numériques intégrées.

Opérateur de reste %

L'opérateur restant `%` calcule le reste après la division de son opérande de partie gauche par son opérande de partie droite.

Reste entier

Pour les opérandes des types entiers, le résultat de `a % b` est la valeur produite par `a - (a / b) * b`. Le signe du reste non zéro est le même que celui de l'opérande de partie gauche, comme l'indique l'exemple suivant :

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Utilisez la méthode Math.DivRem pour calculer à la fois la division entière et les résultats du reste.

Reste à virgule flottante

En ce qui concerne les opérandes `float` et `double`, le résultat de `x % y` pour le `x` et le `y` finis est la valeur `z` afin que

- Le signe de `z`, s'il est différent de zéro, soit identique au signe de `x`.
- La valeur absolue de `z` soit la valeur produite par `|x| - n * |y|`, où `n` représente le plus grand entier possible inférieur ou égal à `|x| / |y|`, et où `|x|` et `|y|` sont les valeurs absolues de `x` et `y`, respectivement.

> **NOTE**
> Cette méthode de calcul du reste est analogue à celle utilisée pour les opérandes entiers, mais différente de la spécification IEEE 754. Si vous avez besoin de l'opération de reste qui est conforme à la spécification IEEE 754, utilisez la Math.IEEERemainder méthode.

Pour plus d'informations sur le comportement de l'opérateur `%` avec des opérandes non finis, consultez la section Opérateur de reste dans la Spécification du langage C#.

Pour les opérandes `decimal`, l'opérateur de reste `%` équivaut à l'opérateur de reste de type System.Decimal.

L'exemple suivant illustre le comportement de l'opérateur de reste pour les opérandes à virgule flottante :

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Opérateur d'addition +

L'opérateur d'addition `+` calcule la somme de ses opérandes :

```
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

Vous pouvez également utiliser l' `+` opérateur pour la concaténation de chaînes et la combinaison de délégués. Pour plus d'informations, consultez l'article `+` `+=` opérateurs et .

Opérateur de soustraction -

L'opérateur de soustraction `-` soustrait son opérande de partie droite de son opérande de partie gauche :

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

Vous pouvez également utiliser l' `-` opérateur pour la suppression d'un délégué. Pour plus d'informations, consultez l'article `-` `-=` opérateurs et .

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```
x op= y
```

équivaut à :

```
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'assignation composée avec des opérateurs arithmétiques :

```
int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

En raison des promotions numériques, le résultat de l'opération `op` risque de ne pas être implicitement convertible en type `T` de `x`. Dans ce cas, si `op` est un opérateur prédéfini et que le résultat de l'opération est explicitement convertible en type `T` de `x`, une expression d'assignation composée de la forme `x op= y` équivaut à `x = (T)(x op y)`, sauf que `x` n'est évalué qu'une seule fois. L'exemple suivant illustre ce comportement :

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

Vous pouvez également utiliser `+=` les `-=` opérateurs et pour vous abonner et vous désabonner d'un événement, respectivement. Pour plus d'informations, consultez Comment s'abonner et annuler l'abonnement à des événements.

Priorité des opérateurs et associativité

La liste suivante présente les opérateurs arithmétiques de la priorité la plus élevée à la plus basse :

- Opérateurs suffixés d'incrémentation `x++` et de décrémentation `x--`
- Opérateurs préfixés d'incrémentation `++x` et de décrémentation `--x` et opérateurs unaires `+` et `-`
- Opérateurs multiplicatifs `*`, `/` et `%`
- Opérateurs additifs `+` et `-`

Les opérateurs arithmétiques binaires sont associatifs sur leur gauche. Autrement dit, les opérateurs de même niveau de priorité sont évalués de gauche à droite.

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la priorité et l'associativité de l'opérateur.

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section priorité d'opérateur de l'article opérateurs c# .

Débordement arithmétique et division par zéro

Quand le résultat d'une opération arithmétique n'est pas compris dans la plage de valeurs finies possibles du type numérique impliqué, le comportement d'un opérateur arithmétique varie selon le type de ses opérandes.

Débordement arithmétique entier

La division d'un entier par zéro lève toujours une exception DivideByZeroException.

En cas de débordement arithmétique entier, un contexte de vérification de débordement, qui peut être vérifié ou non vérifié, contrôle le comportement qui en résulte :

- Dans un contexte vérifié, si le débordement se produit dans une expression constante, une erreur de compilation se produit. Sinon, un OverflowException est levé quand l'opération est effectuée au moment de l'exécution.
- Dans un contexte non vérifié, le résultat est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire.

Avec les instructions vérifiées et non vérifiées, vous pouvez utiliser les opérateurs `checked` et `unchecked` pour contrôler le contexte de vérification de débordement, dans lequel une expression est évaluée :

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

Par défaut, les opérations arithmétiques se produisent dans un contexte *unchecked*.

Débordement arithmétique à virgule flottante

Les opérations arithmétiques avec les types `float` et `double` ne lèvent jamais d'exceptions. Le résultat des opérations arithmétiques avec ces types peut être une des valeurs spéciales qui représentent l'infini et non un nombre :

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

Pour les opérandes du type `decimal`, le débordement arithmétique lève toujours un OverflowException et la division par zéro lève toujours un DivideByZeroException.

Erreurs d'arrondi

En raison des limitations générales de la représentation à virgule flottante des nombres réels et de l'arithmétique à virgule flottante, des erreurs d'arrondi peuvent se produire dans les calculs avec des types à virgule flottante. Autrement dit, le résultat d'une expression peut différer du résultat mathématique attendu. L'exemple suivant illustre plusieurs cas de ce genre :

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.9999999999999999999999999999
```

Pour plus d'informations, consultez la section Notes sur les pages de référence System. double, System. Singleou System. Decimal .

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les `++` opérateurs arithmétiques unaires („ `--` `+` et `-`) et binaires (`*` ,,, `/` `%` `+` et `-`). Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateurs suffixés d'incrémentation et de décrémentation
- Opérateurs préfixés d'incrémentation et de décrémentation
- Opérateur plus unaire
- Opérateur moins unaire
- Opérateur de multiplication
- Opérateur de division
- Remainder (opérateur)
- Opérateur d'addition
- Opérateur de soustraction
- Assignation composée
- Opérateurs vérifiés et non vérifiés
- Promotions numériques

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.Math
- System.MathF
- Valeurs numériques dans .NET

Opérateurs logiques booléens (référence C#)

02/11/2020 • 14 minutes to read • Edit Online

Les opérateurs suivants effectuent des opérations logiques avec des opérandes bool :

- Opérateur unaire `!` (négation logique) .
- Opérateurs binaires `&` (and logique), `|` (or logique)et `^` (or exclusif logique) . qui évaluent toujours les deux opérandes ;
- Opérateurs binaires `&&` (conditionnels and logiques) et `||` (opérateurs logiques conditionnels) . Ces opérateurs n'évaluent l'opérande de partie droite que si nécessaire.

Pour les opérandes des types numériques intégraux, `&` les `|` opérateurs, et `^` effectuent des opérations logiques au niveau du bit. Pour plus d'informations, consultez Opérateurs de bits et de décalage.

L'opérateur de négation logique !

L'opérateur préfixé unaire `!` calcule la négation logique de son opérande. Autrement dit, il produit `true` si l'opérande donne `false` et `false` si l'opérande donne `true` :

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

À compter de C# 8,0, l'opérateur postfix unaire `!` est l' opérateur null-indulgent avec.

Opérateur AND logique&

L'opérateur `&` calcule le AND logique de ses opérandes. Le résultat de `x & y` est `true` si `x` et `y` prennent la valeur `true`. Sinon, le résultat est `false`.

L' `&` opérateur évalue les deux opérandes même si l'opérande de gauche est évalué à `false` , de sorte que le résultat de l'opération est `false` indépendamment de la valeur de l'opérande de droite.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `&` est un appel de méthode, effectué indépendamment de la valeur de l'opérande de partie gauche :

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur AND logique conditionnel `&&` calcule également le AND logique de ses opérandes, mais n'évalue pas l'opérande de partie droite si l'opérande de partie gauche donne la valeur de `false`.

Pour les opérandes des types numériques intégraux, l' `&` opérateur calcule le and logique au niveau du bit de ses opérandes. L'opérateur unaire `&` est l'opérateur address-of.

L'opérateur OR exclusif logique ^

L'opérateur `^` calcule le OR exclusif logique, également appelé XOR logique, de ses opérandes. Le résultat de `x ^ y` est `true` si `x` donne `true` et `y` donne `false`, ou `x` donne `false` et `y` donne `true`. Sinon, le résultat est `false`. Autrement dit, pour les opérandes `bool`, l'opérateur `^` calcule le même résultat que l'opérateur d'inégalité `!=`.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

Pour les opérandes des types numériques intégraux, l' `^` opérateur calcule l' or exclusif logique au niveau du bit de ses opérandes.

L'opérateur OU logique |

L'opérateur `|` calcule le OR logique de ses opérandes. Le résultat de `x | y` est `true` si `x` ou `y` prend la valeur `true`. Sinon, le résultat est `false`.

L' `|` opérateur évalue les deux opérandes même si l'opérande de gauche est évalué à `true` , de sorte que le résultat de l'opération est `true` indépendamment de la valeur de l'opérande de droite.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `|` est un appel de méthode, effectué indépendamment de la valeur de l'opérande de partie gauche :

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur OR logique conditionnel `||` calcule également le OR logique de ses opérandes, mais n'évalue pas l'opérande de partie droite si l'opérande de partie gauche donne la valeur de `true`.

Pour les opérandes des types numériques intégraux, l' `|` opérateur calcule le or logique au niveau du bit de ses opérandes.

Opérateur logique AND conditionnel&&

L'opérateur AND logique conditionnel `&&`, également appelé opérateur AND logique de « court-circuit », calcule le AND logique de ses opérandes. Le résultat de `x && y` est `true` si `x` et `y` prennent la valeur `true`. Sinon, le résultat est `false`. Si `x` donne la valeur de `false`, `y` n'est pas évalué.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `&&` est un appel de méthode, non effectué si l'opérande de partie gauche donne la valeur de `false` :

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur AND logique `&` calcule également le AND logique de ses opérandes, mais évalue toujours les deux opérandes.

L'opérateur OR logique conditionnel ||

L'opérateur OR logique conditionnel `||`, également appelé opérateur OR logique de « court-circuit », calcule le OR logique de ses opérandes. Le résultat de `x || y` est `true` si `x` ou `y` prend la valeur `true`. Sinon, le résultat est `false`. Si `x` donne la valeur de `true`, `y` n'est pas évalué.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `||` est un appel de méthode, non effectué si l'opérande de partie gauche donne la valeur de `true` :

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L' opérateur OR logique `|` calcule également le or logique de ses opérandes, mais évalue toujours les deux opérandes.

Les opérateurs logiques booléens Nullable

Pour les `bool?` opérandes, les opérateurs `&` (logique and) et `|` (logique or) prennent en charge la logique à

trois valeurs comme suit :

- L' `&` opérateur produit `true` uniquement si ses opérandes ont tous les deux la valeur `true` . Si ou a la `x` `y` valeur `false` , `x & y` produit `false` (même si un autre opérande a la valeur `null`). Sinon, le résultat de `x & y` est `null` .
- L' `|` opérateur produit `false` uniquement si ses opérandes ont tous les deux la valeur `false` . Si ou a la `x` `y` valeur `true` , `x | y` produit `true` (même si un autre opérande a la valeur `null`). Sinon, le résultat de `x | y` est `null` .

Le tableau suivant présente cette sémantique :

X	Y	X&Y	X|Y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Le comportement de ces opérateurs diffère du comportement classique des opérateurs avec des types valeur Nullable. En règle générale, un opérateur défini pour les opérandes d'un type valeur peut être également utilisé avec des opérandes du type valeur Nullable correspondant. Un tel opérateur produit `null` si l'un de ses opérandes a la valeur `null` . Toutefois, les `&` `|` opérateurs et peuvent produire des valeurs non null, même si l'un des opérandes a la valeur `null` . Pour plus d'informations sur le comportement de l'opérateur avec les types valeur Nullable, consultez la section opérateurs levés de l'article types valeur Nullable .

Vous pouvez également utiliser les `!` `^` opérateurs et avec des `bool?` opérandes, comme le montre l'exemple suivant :

```
bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

Les opérateurs logiques conditionnels `&&` et `||` ne prennent pas en charge les `bool?` opérandes.

Assignation composée

Pour un opérateur binaire `op` , une expression d'assignation composée du formulaire

```
x op= y
```

équivaut à :

```
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

Les opérateurs `&`, `|` et `^` prennent en charge l'assignation composée, comme le montre l'exemple suivant :

```
bool test = true;
test &= false;
Console.WriteLine(test);  // output: False

test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

NOTE

Les opérateurs logiques conditionnels `&&` et `||` ne prennent pas en charge l'assignation composée.

Précédence des opérateurs

La liste suivante présente les opérateurs logiques par ordre de précédence, de la plus élevée à la plus basse :

- Opérateur de négation logique `!`
- L'opérateur AND logique `&`
- Opérateur OR exclusif logique `^`
- Opérateur OR logique `|`
- Opérateur AND logique conditionnel `&&`
- Opérateur OR logique conditionnel `||`

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs :

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section priorité d'opérateur de l'article opérateurs c# .

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les `!` opérateurs,, `&` `|` et `^` . Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

Un type défini par l'utilisateur ne peut pas surcharger les opérateurs logiques conditionnels `&&` et `||`. Toutefois, si un type défini par l'utilisateur surcharge les opérateurs true et false et l'opérateur `&` ou `|` d'une certaine manière, l'opération `&&` ou `||` peut être évaluée respectivement pour les opérandes de ce type. Pour plus d'informations, consultez la section Opérateurs logiques conditionnels définis par l'utilisateur de la spécification du langage C#.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateur de négation logique
- Opérateurs logiques
- Opérateurs logiques conditionnels
- Assignation composée

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs de bits et de décalage

Opérateurs au niveau du bit et opérateurs de décalage (référence C#)

02/11/2020 • 14 minutes to read • Edit Online

Les opérateurs suivants effectuent des opérations de bits ou de décalage avec les opérandes des types numériques intégraux ou le type char :

- Opérateur unaire `~` (complément de bits)
- Opérateurs de décalage binaire `<<` (décalage vers la gauche) et `>>` (décalage vers la droite)
- Opérateurs binaires `&` (and logique), `|` (or logique)et (opérateur or `^` exclusif logique)

Ces opérateurs sont définis pour les types `int`, `uint`, `long` et `ulong`. Lorsque les deux opérandes sont d'autres types intégraux (`sbyte`, `byte`, `short`, `ushort` ou `char`), leurs valeurs sont converties en valeurs de type `int`, qui est également le type de résultat d'une opération. Lorsque les opérandes sont de différents types intégraux, leurs valeurs sont converties vers le type intégral le plus proche. Pour plus d'informations, consultez la section Promotions numériques de la spécification du langage C#.

Les `&` `|` opérateurs, et `^` sont également définis pour les opérandes du `bool` type. Pour plus d'informations, consultez Opérateurs logiques booléens.

Les opérations de décalage et au niveau du bit ne provoquent jamais de dépassements de capacité et donnent les mêmes résultats dans des contextes checked et unchecked.

Opérateur de complément de bits ~

L'opérateur `~` produit un complément de bits de son opérande en inversant chaque bit :

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

Vous pouvez également utiliser le symbole `~` pour déclarer des finaliseurs. Pour plus d'informations, consultez Finaliseurs.

Opérateur de décalage gauche <<

L' `<<` opérateur décale son opérande de gauche vers la gauche du nombre de bits défini par son opérande de droite.

L'opération de décalage gauche supprime les bits d'ordre supérieur qui sont en dehors de la plage du type de résultat, et définit les positions de bits vides d'ordre inférieur sur zéro, comme le montre l'exemple suivant :

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Étant donné que les opérateurs de décalage sont définis uniquement pour les types `int`, `uint`, `long` et `ulong`, le résultat d'une opération contient toujours au moins 32 bits. Si l'opérande de partie gauche est d'un autre type intégral (`sbyte`, `byte`, `short`, `ushort` ou `char`), sa valeur est convertie en type `int`, comme l'indique l'exemple suivant :

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Pour plus d'informations sur la façon dont l'opérande de partie droite de l'opérateur `<<` définit la valeur de décalage, consultez la section Valeur de décalage des opérateurs de décalage.

Opérateur de décalage vers la droite >>

L' `>>` opérateur décale son opérande de gauche vers la droite du nombre de bits défini par son opérande de droite.

L'opération de décalage vers la droite ignore les bits d'ordre inférieur, comme le montre l'exemple suivant :

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

Les positions de bits vides d'ordre supérieur sont définies en fonction du type de l'opérande de partie gauche comme suit :

- Si l'opérande de gauche est de type `int` ou `long`, l'opérateur de décalage vers la droite effectue un décalage *arithmétique* : la valeur du bit le plus significatif (le bit de signe) de l'opérande de gauche est propagée aux positions de bits vides de poids fort. Autrement dit, les positions de bits vides d'ordre supérieur sont définies sur zéro si l'opérande de partir gauche n'est pas négatif et sur un s'il est négatif.

```
int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:  11110000000000000000000000000000
```

- Si l'opérande de gauche est de type `uint` ou `ulong`, l'opérateur de décalage vers la droite effectue un décalage *logique* : les positions de bits vides de poids fort ont toujours la valeur zéro.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:     10000000000000000000000000000
```

Pour plus d'informations sur la façon dont l'opérande de partie droite de l'opérateur `>>` définit la valeur de décalage, consultez la section Valeur de décalage des opérateurs de décalage.

Opérateur AND logique &

L' `&` opérateur calcule le and logique au niveau du bit de ses opérandes intégraux :

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

Pour les `bool` opérandes, l' `&` opérateur calcule le and logique de ses opérandes. L'opérateur unaire `&` est l'opérateur address-of.

L'opérateur OR exclusif logique ^

L' `^` opérateur calcule l'or exclusif logique au niveau du bit, également appelé XOR logique au niveau du bit, de ses opérandes intégraux :

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

Pour les `bool` opérandes, l' `^` opérateur calcule l' or exclusif logique de ses opérandes.

L'opérateur OU logique |

L' `|` opérateur calcule le or logique au niveau du bit de ses opérandes intégraux :

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Pour les `bool` opérandes, l' `|` opérateur calcule le or logique de ses opérandes.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```
x op= y
```

équivaut à :

```
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant montre l'utilisation de l'assignation composée avec des opérateurs de décalage et des opérateurs au niveau du bit :

```
uint a = 0b_1111_1000;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a |= 0b_0011_0001;
Display(a);  // output: 10111001

a ^= 0b_1000_0000;
Display(a);  // output:   111001

a <<= 2;
Display(a);  // output: 11100100

a >>= 4;
Display(a);  // output:     1110

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");
```

En raison des promotions numériques, le résultat de l'opération `op` risque de ne pas être implicitement convertible en type `T` de `x`. Dans ce cas, si `op` est un opérateur prédéfini et que le résultat de l'opération est explicitement convertible en type `T` de `x`, une expression d'assignation composée de la forme `x op= y` équivaut à `x = (T)(x op y)`, sauf que `x` n'est évalué qu'une seule fois. L'exemple suivant illustre ce comportement :

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output: 1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Précédence des opérateurs

La liste suivante présente les opérateurs au niveau du bit et les opérateurs de décalage par ordre de précédence, de la plus élevée à la plus basse :

- Opérateur de complément de bits `~`
- Opérateurs de décalage `<<` et `>>`
- L'opérateur AND logique `&`
- Opérateur OR exclusif logique `^`
- Opérateur OR logique `|`

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs :

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section priorité d'opérateur de l'article opérateurs c# .

Valeur de décalage des opérateurs de décalage

Pour les opérateurs de décalage `<<` et `>>` , le type de l'opérande de droite doit être `int` ou un type qui a une conversion numérique implicite prédéfinie en `int` .

Pour les expressions `x << count` et `x >> count`, la valeur réelle du décalage varie selon le type de `x` :

- Si le type de `x` est `int` ou `uint` , le nombre de décalages est défini par les *cinq* bits de poids faible de l'opérande de droite. La valeur de décalage est donc calculée à partir de `count & 0x1F` (ou de `count & 0b_1_1111`).
- Si le type de `x` est `long` ou `ulong` , le nombre de décalages est défini par les *six* bits de poids faible de l'opérande de droite. La valeur de décalage est donc calculée à partir de `count & 0x3F` (ou de `count & 0b_11_1111`).

L'exemple suivant illustre ce comportement :

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

> **NOTE**
> Comme le montre l'exemple précédent, le résultat d'une opération de décalage peut être différent de zéro même si la valeur de l'opérande de droite est supérieure au nombre de bits dans l'opérande de gauche.

Opérateurs logiques d'énumération

Les `~` opérateurs,, `&` `|` et `^` sont également pris en charge par n'importe quel type énumération . Pour les opérandes du même type d'énumération, une opération logique est effectuée sur les valeurs correspondantes du type intégral sous-jacent. Par exemple, pour tout `x` et `y` d'un type énumération `T` avec un type sous-jacent `U`, l'expression `x & y` produit le même résultat que l'expression `(T)((U)x & (U)y)`.

En général, vous utilisez des opérateurs logiques au niveau du bit avec un type d'énumération défini avec l'attribut Flags . Pour plus d'informations, consultez la section Types énumération comme indicateurs binaires de l'articleTypes énumération.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs,,,, `~` `<<` `>>` `&` `|` et `^` . Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

Si un type `T` défini par l'utilisateur surcharge l'opérateur `<<` ou `>>`, l'opérande de partie gauche doit être de type `T` et l'opérande de partie droite de type `int`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateur de complément de bits
- Opérateurs de décalage
- Opérateurs logiques
- Assignation composée
- Promotions numériques

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs logiques booléens

Opérateurs d'égalité (référence C#)

02/11/2020 • 7 minutes to read • Edit Online

Les opérateurs `==` (égalité) et `!=` (inégalité) vérifient si leurs opérandes sont égaux ou non.

Opérateur d'égalité ==

L'opérateur d'égalité `==` retourne `true` si ses opérandes sont égaux, `false` dans le cas contraire.

Égalité de types valeur

Les opérandes des types valeur intégrés sont égaux si leurs valeurs sont égales :

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

NOTE

Pour les `==` opérateurs, `<` , `>` , `<=` et `>=` , si l'un des opérandes n'est pas un nombre (Double.NaN ou Single.NaN), le résultat de l'opération est `false` . Cela signifie que la valeur `NaN` n'est ni supérieure à, ni inférieure à, ni égale à n'importe quelle autre valeur `double` (ou `float`), y compris `NaN` . Pour plus d'informations et des exemples, consultez l'article de référence Double.NaN ou Single.NaN.

Deux opérandes du même type enum sont égaux si les valeurs correspondantes du type intégral sous-jacent sont égales.

Par défaut, les types struct définis par l'utilisateur ne prennent pas en charge l'opérateur `==` . Pour prendre en charge l'opérateur `==` , un struct défini par l'utilisateur doit le surcharger.

À compter de C# 7.3, les opérateurs `==` et `!=` sont pris en charge par les tuples C#. Pour plus d'informations, consultez la section égalité des tuples de l'article types de tuple .

Égalité de types référence

Par défaut, deux opérandes de type référence non-enregistrement sont égaux s'ils font référence au même objet :

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

Comme le montre l'exemple, les types référence définis par l'utilisateur prennent en charge l'opérateur `==` par défaut. Un type référence peut toutefois surcharger l'opérateur `==`. Si un type référence surcharge l'opérateur `==`, utilisez la méthode Object.ReferenceEquals pour vérifier si deux références de ce type font référence au même objet.

Égalité des types d'enregistrements

Disponible en C# 9,0 et versions ultérieures, les types d'enregistrements prennent en charge les `==` `!=` opérateurs et qui, par défaut, fournissent une sémantique d'égalité des valeurs. Autrement dit, deux opérandes d'enregistrement sont égaux lorsqu'ils sont ou que les `null` valeurs correspondantes de tous les champs et les propriétés implémentées automatiquement sont égales.

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2);  // output: False
        Console.WriteLine(p1 == p3);  // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2);  // output: False
    }
}
```

Comme le montre l'exemple précédent, dans le cas de membres de type référence non-enregistrement, leurs valeurs de référence sont comparées, et non les instances référencées.

Égalité de chaîne

Deux opérandes string sont égaux lorsque les deux sont `null` ou lorsque les deux instances de chaîne sont de même longueur et ont des caractères identiques dans chaque position de caractère :

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

Il s'agit d'une comparaison ordinale respectant la casse. Pour plus d'informations sur la comparaison de chaînes, consultez Comment comparer des chaînes en C#.

Égalité de délégué

Deux opérandes de délégué du même type de runtime sont égaux si les deux sont `null` ou si leurs listes d'appel sont de même longueur et ont des entrées égales dans chaque position :

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

Pour plus d'informations, consultez la section Opérateurs d'égalité de délégué de la spécification du langage C#.

Les délégués qui sont produits à partir de l'évaluation d'expressions lambda identiques sémantiquement ne sont pas égaux, comme l'indique l'exemple suivant :

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

Opérateur d'inégalité !=

L'opérateur d'inégalité `!=` retourne `true` si ses opérandes ne sont pas égaux, `false` dans le cas contraire. Pour les opérandes des types intégrés, l'expression `x != y` produit le même résultat que l'expression `!(x == y)`. Pour plus d'informations sur l'égalité des types, consultez la section Opérateur d'égalité.

L'exemple suivant illustre l'utilisation de l'opérateur `!=` :

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs `==` et `!=`. Si un type surcharge l'un des deux opérateurs, il doit également surcharger l'autre.

Un type d'enregistrement ne peut pas surcharger explicitement les `==` `!=` opérateurs et. Si vous avez besoin de modifier le comportement des `==` `!=` opérateurs et pour le type d'enregistrement `T` , implémentez la IEquatable<T>.Equals méthode avec la signature suivante :

```
public virtual bool Equals(T? other);
```

spécification du langage C#

Pour plus d'informations, consultez la section Opérateurs relationnels et de test de type de la spécification du langage C#.

Pour plus d'informations sur l'égalité des types d'enregistrements, consultez la section membres d'égalité de la remarque relative à la proposition de la fonctionnalité enregistrements.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- Comparaisons d'égalité
- Opérateurs de comparaison

Opérateurs de comparaison (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Les opérateurs `<` (inférieur à), `>` (supérieurà), `<=` (inférieur ou égal à)et `>=` (supérieur ou égal à) , également connus sous le nom d'opérateurs relationnels, comparent leurs opérandes. Ces opérateurs sont pris en charge par tous les types numériques intégraux et à virgule flottante .

> **NOTE**
> Pour les opérateurs `==` , `<` , `>` , `<=` et `>=` , si un des opérandes n'est pas un nombre (Double.NaN ou Single.NaN), le résultat de l'opération est `false` . Cela signifie que la valeur `NaN` n'est ni supérieure à, ni inférieure à, ni égale à n'importe quelle autre valeur `double` (ou `float`), y compris `NaN` . Pour plus d'informations et des exemples, consultez l'article de référence Double.NaN ou Single.NaN.

Le type char prend également en charge les opérateurs de comparaison. Dans le cas des `char` opérandes, les codes de caractère correspondants sont comparés.

Les types d'énumération prennent également en charge les opérateurs de comparaison. Pour les opérandes du même type enum, les valeurs correspondantes du type intégral sous-jacent sont comparées.

Les `==` `!=` opérateurs et vérifient si leurs opérandes sont égaux ou non.

Opérateur Inférieur à <

L'opérateur `<` retourne `true` si son opérande de partie gauche est inférieur à son opérande de partie droite, `false` dans le cas contraire :

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Opérateur Supérieur à >

L'opérateur `>` retourne `true` si son opérande de partie gauche est supérieur à son opérande de partie droite, `false` dans le cas contraire :

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Opérateur Inférieur ou égal à <=

L'opérateur `<=` retourne `true` si son opérande de partie gauche est inférieur ou égal à son opérande de partie droite, `false` dans le cas contraire :

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Opérateur Supérieur ou égal à >=

L'opérateur `>=` retourne `true` si son opérande de partie gauche est supérieur ou égal à son opérande de partie droite, `false` dans le cas contraire :

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les `<` opérateurs,, `>` `<=` et `>=` .

Si un type surcharge un des opérateurs `<` ou `>`, il doit surcharger à la fois `<` et `>`. Si un type surcharge un des opérateurs `<=` ou `>=`, il doit surcharger à la fois `<=` et `>=`.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateurs relationnels et de test de type de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.IComparable<T>
- Opérateurs d'égalité

Opérateurs et expressions d'accès aux membres (référence C#)

16/12/2020 • 14 minutes to read • Edit Online

Vous pouvez utiliser les opérateurs et expressions suivants lorsque vous accédez à un membre de type :

- `.` (accès aux membres): pour accéder à un membre d'un espace de noms ou d'un type
- `[]` (élément de tableau ou accès à l'indexeur): pour accéder à un élément de tableau ou à un indexeur de type
- `?.` et `?[]` (opérateurs conditionnels null): pour effectuer une opération d'accès de membre ou d'élément uniquement si un opérande n'est pas null
- `()` (appel): pour appeler une méthode accédée ou appeler un délégué
- `^` (index à partir de la fin): pour indiquer que la position de l'élément est à partir de la fin d'une séquence
- `..` (plage): pour spécifier une plage d'index que vous pouvez utiliser pour obtenir une plage d'éléments de séquence

Expression d'accès au membre.

Le jeton `.` sert à accéder à l'un des membres d'un espace de noms ou d'un type, comme le montrent les exemples suivants :

- Utilisez `.` pour accéder à un espace de noms imbriqué dans un espace de noms, comme le montre l'exemple suivant d'une `using` directive :

  ```
  using System.Collections.Generic;
  ```

- Utilisez `.` pour former un *nom qualifié* permettant d'accéder à un type dans un espace de noms, comme le montre le code suivant :

  ```
  System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
  ```

  Utilisez une `using` directive pour rendre l'utilisation des noms qualifiés facultative.

- Utilisez `.` pour accéder aux membres de type, statiques et non statiques, comme le montre le code suivant :

  ```
  var constants = new List<double>();
  constants.Add(Math.PI);
  constants.Add(Math.E);
  Console.WriteLine($"{constants.Count} values to show:");
  Console.WriteLine(string.Join(", ", constants));
  // Output:
  // 2 values to show:
  // 3.14159265358979, 2.71828182845905
  ```

Vous pouvez également utiliser `.` pour accéder à une méthode d'extension.

Opérateur d'indexeur []

Les crochets, `[]`, sont généralement utilisés pour l'accès à un élément tableau, indexeur ou pointeur.

Accès aux tableaux

L'exemple suivant montre comment accéder à des éléments tableau :

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

Si un index de tableau est en dehors des limites de la dimension correspondante d'un tableau, une IndexOutOfRangeException est levée.

Comme le montre l'exemple précédent, vous utilisez également des crochets quand vous déclarez un type tableau ou instanciez une instance de tableau.

Pour plus d'informations sur les tableaux, consultez Tableaux.

Accès aux indexeurs

L'exemple suivant utilise le Dictionary<TKey,TValue> type .net pour illustrer l'accès à l'indexeur :

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

Les indexeurs vous permettent d'indexer des instances d'un type défini par l'utilisateur en procédant de la même façon que pour l'indexation de tableau. Contrairement aux index de tableau, qui doivent être des entiers, les paramètres de l'indexeur peuvent être déclarés comme n'importe quel type.

Pour plus d'informations sur les indexeurs, consultez Indexeurs.

Autres utilisations de []

Pour plus d'informations concernant l'accès à l'élément de pointeur, consultez la section Opérateur d'accès à l'élément de pointeur [] de l'article Opérateurs associés au pointeur.

Vous utilisez également des crochets pour spécifier des attributs :

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Opérateurs conditionnels Null ?. et ?[]

Disponible en C# 6 et versions ultérieures, un opérateur conditionnel null applique un accès de membre, `?.` ou d'accès à l' élément, `?[]` , à son opérande uniquement si cet opérande a la valeur non NULL ; sinon, il retourne `null` . C'est

- Si `a` prend la valeur `null` , le résultat de `a?.x` ou `a?[x]` est `null` .

- Si `a` prend la valeur non null, le résultat de `a?.x` ou `a?[x]` est le même que le résultat de `a.x` ou `a[x]`, respectivement.

> **NOTE**
> Si `a.x` ou `a[x]` lèvent une exception, `a?.x` ou `a?[x]` lèvent la même exception pour la valeur non null `a`. Par exemple, si `a` est une instance de tableau non null et `x` se trouve en dehors des limites de `a`, `a?[x]` lèvera un IndexOutOfRangeException.

Les opérateurs conditionnels Null ont un effet de court-circuit. Autrement dit, si une opération dans une chaîne d'opérations d'accès au membre ou à l'élément conditionnelles retourne une valeur `null`, le reste de la chaîne ne s'exécute pas. Dans l'exemple suivant, `B` n'est pas évalué si `A` prend la valeur `null` et `C` n'est pas évalué si `A` ou `B` prend la valeur `null` :

```
A?.B?.Do(C);
A?.B?[C];
```

L'exemple suivant illustre l'utilisation des opérateurs `?.` et `?[]` :

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

L'exemple précédent utilise également l' opérateur `??` de fusion Null pour spécifier une autre expression à évaluer si le résultat d'une opération conditionnelle null est `null`.

Si `a.x` ou `a[x]` est un type valeur n'acceptant pas `T` les valeurs NULL, `a?.x` ou `a?[x]` est du type valeur Nullable correspondant `T?`. Si vous avez besoin d'une expression de type `T`, appliquez l'opérateur de fusion Null `??` à une expression conditionnelle null, comme le montre l'exemple suivant :

```
int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 }));  // output: 7
```

Dans l'exemple précédent, si vous n'utilisez pas l' `??` opérateur, `numbers?.Length < 2` prend la valeur `false` lorsque `numbers` est `null` .

L'opérateur d'accès aux membres conditionnels null `?.` est également appelé l'opérateur Elvis.

Appel de délégué thread-safe

Utilisez l'opérateur `?.` pour vérifier si un délégué est non Null et l'appeler de manière thread-safe (par exemple, quand vous déclenchez un événement), comme illustré dans le code suivant :

```
PropertyChanged?.Invoke(…)
```

Ce code est équivalent au code suivant que vous utiliseriez dans C# 5 ou une version antérieure :

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

C'est une façon thread-safe de s'assurer que seule une valeur non null `handler` est appelée. Étant donné que les instances de délégué sont immuables, aucun thread ne peut modifier l'objet référencé par la `handler` variable locale. En particulier, si le code exécuté par un autre thread annule son abonnement à l' `PropertyChanged` événement et `PropertyChanged` devient `null` avant l'appel de `handler` , l'objet référencé par `handler` reste inchangé. L' `?.` opérateur évalue son opérande de gauche un peu plus d'une fois, ce qui garantit qu'il ne peut pas être modifié `null` après avoir été vérifié comme non null.

Expression d'appel ()

Utilisez des parenthèses, `()`, pour appeler une méthode ou un délégué.

L'exemple suivant montre comment appeler une méthode, avec ou sans arguments, et un délégué :

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

Vous utilisez également des parenthèses quand vous appelez un constructeur avec l'opérateur `new`.

Autres utilisations de ()

Vous utilisez également des parenthèses pour ajuster l'ordre dans lequel évaluer les opérations dans une expression. Pour plus d'informations, consultez Opérateurs C#.

Les expressions cast, qui effectuent des conversions de type explicites, utilisent aussi des parenthèses.

Index de fin d'opérateur ^

Disponible en C# 8,0 et versions ultérieures, l' `^` opérateur indique la position de l'élément à partir de la fin d'une séquence. Pour une séquence de longueur `length` , `^n` pointe vers l'élément avec décalage `length - n` à partir du début d'une séquence. Par exemple, `^1` pointe vers le dernier élément d'une séquence et `^length` pointe vers le premier élément d'une séquence.

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last);  // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

Comme le montre l'exemple précédent, expression `^e` est du System.Index type. Dans Expression `^e` , le résultat de `e` doit être implicitement convertible en `int` .

Vous pouvez également utiliser l' `^` opérateur avec l' opérateur Range pour créer une plage d'index. Pour plus d'informations, consultez index et plages.

Opérateur de plage..

Disponible en C# 8,0 et versions ultérieures, l' `..` opérateur spécifie le début et la fin d'une plage d'index comme opérandes. L'opérande de gauche est un début *inclusif* d'une plage. L'opérande de droite est une extrémité *exclusive* d'une plage. L'un ou l'autre des opérandes peut être un index à partir du début ou de la fin d'une séquence, comme le montre l'exemple suivant :

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

Comme le montre l'exemple précédent, expression `a..b` est du System.Range type. Dans Expression `a..b` , les

résultats de `a` et `b` doivent être implicitement convertibles en `int` ou Index .

Vous pouvez omettre l'un des opérandes de l' `..` opérateur pour obtenir une plage ouverte :

- `a..` équivaut à `a..^0`
- `..b` équivaut à `0..b`
- `..` équivaut à `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

Pour plus d'informations, consultez index et plages.

Capacité de surcharge de l'opérateur

Les `.` opérateurs,, `()` `^` et `..` ne peuvent pas être surchargés. L'opérateur `[]` est également considéré comme un opérateur non surchargeable. Utilisez des indexeurs pour prendre en charge l'indexation avec des types définis par l'utilisateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Accès au membre
- Accès aux éléments
- Opérateur conditionnel Null
- Expressions d'appels

Pour plus d'informations sur les index et les plages, consultez la Remarque relativeà la proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- ?? (opérateur de fusion Null)
- :: (opérateur)

Opérateurs de test de type et expression de cast (référence C#)

02/11/2020 • 11 minutes to read • Edit Online

Vous pouvez utiliser les opérateurs et expressions suivants pour effectuer une vérification de type ou une conversion de type :

- Opérateur is : pour vérifier si le type de runtime d'une expression est compatible avec un type donné
- Opérateur as : pour convertir explicitement une expression en un type donné si son type de runtime est compatible avec ce type
- expression de cast: pour effectuer une conversion explicite
- Opérateur typeof : pour obtenir l'instance de System.Type pour un type

Opérateur is

L'opérateur `is` vérifie si le type de runtime du résultat d'une expression est compatible avec un type donné. À compter de C# 7,0, l' `is` opérateur teste également un résultat d'expression par rapport à un modèle.

L'expression avec l'opérateur de test de type `is` a la forme suivante

```
E is T
```

où `E` est une expression qui retourne une valeur et `T` est le nom d'un type ou un d'un paramètre de type. `E` ne peut pas être une méthode anonyme ou une expression lambda.

L'expression `E is T` retourne `true` si le résultat de `E` n'est pas null et peut être converti en type `T` par une conversion de référence, une conversion boxing ou une conversion unboxing ; sinon, elle retourne `false`. L'opérateur `is` ne considère pas les conversions définies par l'utilisateur.

L'exemple suivant montre que l'opérateur `is` retourne `true` si le type de runtime d'une expression dérive d'un type donné, autrement dit, s'il existe une conversion de référence entre les types :

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

L'exemple suivant montre que l' `is` opérateur prend en compte les conversions boxing et unboxing, mais ne prend pas en compte les conversions numériques:

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

Pour plus d'informations sur les conversions C#, consultez le chapitre Conversions de la Spécification du langage C#.

Test de type avec des critères spéciaux

À compter de C# 7,0, l' `is` opérateur teste également un résultat d'expression par rapport à un modèle. En particulier, il prend en charge le modèle de type sous la forme suivante :

```
E is T v
```

où `E` est une expression qui retourne une valeur, `T` est le nom d'un type ou un d'un paramètre de type et `v` est une nouvelle variable locale de type `T`. Si le résultat de `E` n'est pas null et peut être converti en `T` par conversion de référence, boxing ou unboxing, l'expression `E is T v` retourne `true` et la valeur convertie du résultat de `E` est affectée à la variable `v`.

L'exemple suivant illustre l'utilisation de l'opérateur `is` avec le modèle de type :

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

Pour plus d'informations sur le modèle de type et les autres modèles pris en charge, consultez Critères spéciaux avec is.

opérateur as

L'opérateur `as` convertit explicitement le résultat d'une expression en un type de valeur de référence ou nullable. Si la conversion n'est pas possible, l'opérateur `as` retourne `null`. Contrairement à une expression de cast, l' `as` opérateur ne lève jamais d'exception.

L'expression de la forme

```
E as T
```

où `E` est une expression qui retourne une valeur et `T` est le nom d'un type ou un d'un paramètre de type, avec le même résultat que

```
E is T ? (T)(E) : (T)null
```

sauf que `E` n'est évalué qu'une seule fois.

L'opérateur `as` envisage uniquement les conversions de référence, nullable, boxing et unboxing. Vous ne pouvez pas utiliser l'opérateur `as` pour effectuer une conversion définie par l'utilisateur. Pour ce faire, utilisez une

expression de cast.

L'exemple suivant illustre l'utilisation de l'opérateur `as` :

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  // output: 40
}
```

NOTE

Comme le montre l'exemple précédent, vous devez comparer le résultat de l'expression `as` avec `null` pour vérifier si la conversion a réussi. À compter de C# 7,0, vous pouvez utiliser l' opérateur is pour tester si la conversion aboutit et, si elle aboutit, assigner son résultat à une nouvelle variable.

Expression Cast

Une expression cast de la forme `(T)E` effectue une conversion explicite du résultat de l'expression `E` en type `T`. S'il n'existe aucune conversion explicite du type de `E` en type `T`, une erreur de compilation se produit. Au moment de l'exécution, une conversion explicite peut ne pas réussir, et une expression cast peut lever une exception.

L'exemple suivant montre des conversions numériques et de référence explicites :

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

Pour plus d'informations sur les conversions explicites prises en charge, consultez la section Conversions explicites de la Spécification du langage C#. Pour plus d'informations sur la façon de définir une conversion de type explicite ou implicite personnalisée, consultez Opérateurs de conversion définie par l'utilisateur.

Autres utilisations de ()

Vous pouvez aussi utiliser des parenthèses pour appeler une méthode ou un délégué.

Une autre utilisation des parenthèses est d'ajuster l'ordre dans lequel évaluer les opérations dans une expression. Pour plus d'informations, consultez Opérateurs C#.

Opérateur typeof

L'opérateur `typeof` obtient l'instance System.Type pour un type. L'argument de l'opérateur `typeof` doit avoir le nom d'un type ou d'un paramètre de type, comme le montre l'exemple suivant :

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

Vous pouvez également utiliser l' `typeof` opérateur avec des types génériques indépendants. Le nom d'un type générique indépendant doit contenir le nombre approprié de virgules, à savoir une de moins que le nombre de paramètres de type. L'exemple suivant illustre l'utilisation de l'opérateur `typeof` avec un type générique indépendant :

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

Une expression ne peut pas être un argument de l'opérateur `typeof`. Pour obtenir l' System.Type instance du type au moment de l'exécution d'un résultat d'expression, utilisez la Object.GetType méthode.

Test de type avec l'opérateur `typeof`

Utilisez l'opérateur `typeof` pour vérifier si le type de runtime du résultat de l'expression correspond exactement à un type donné. L'exemple suivant illustre la différence entre la vérification des types effectuée avec l'opérateur `typeof` l'opérateur is :

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

Capacité de surcharge de l'opérateur

Les `is` `as` opérateurs, et `typeof` ne peuvent pas être surchargés.

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur `()`, mais vous pouvez définir des conversions de type personnalisées qui peuvent être effectuées par une expression cast. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- L'opérateur is
- L'opérateur as
- Expressions cast
- Opérateur typeof

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Comment effectuer un cast en toute sécurité à l'aide de critères spéciaux et des opérateurs is et As
- Génériques en .NET

Opérateurs de conversion définie par l'utilisateur (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Un type défini par l'utilisateur peut définir une conversion implicite ou explicite personnalisée depuis ou vers un autre type.

Les conversions implicites ne nécessitent pas une syntaxe spéciale à appeler et peuvent se produire dans diverses situations, par exemple, dans les appels de méthodes et les affectations. Les conversions implicites C# prédéfinies fonctionnent toujours et ne lèvent jamais d'exception. Les conversions implicites définies par l'utilisateur doivent aussi se comporter de cette façon. Si une conversion personnalisée peut lever une exception ou perdre des informations, définissez-la comme conversion explicite.

Les conversions définies par l'utilisateur ne sont pas prises en compte par les opérateurs is et as. Utilisez une expression de cast pour appeler une conversion explicite définie par l'utilisateur.

Utilisez les mots clés `operator` et `implicit` ou `explicit` pour définir une conversion implicite ou explicite, respectivement. Le type qui définit une conversion doit être un type source ou un type cible de cette conversion. Une conversion entre deux types définis par l'utilisateur peut être définie dans l'un ou l'autre des deux types.

L'exemple suivant montre comment définir une conversion implicite et explicite :

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

Utilisez également le mot clé `operator` pour surcharger un opérateur C# prédéfini. Pour plus d'informations, consultez Surcharge d'opérateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateurs de conversion
- Conversions définies par l'utilisateur
- Conversions implicites
- Conversions explicites

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Surcharge d'opérateur
- Opérateurs de conversion et de test de type
- Cast et conversion de types
- Instructions de conception-opérateurs de conversion
- Conversions explicites définies par l'utilisateur chaînées en C#

Opérateurs associés au pointeur (référence C#)

02/11/2020 • 13 minutes to read • Edit Online

Vous pouvez utiliser les opérateurs suivants avec les pointeurs :

- Opérateur unaire `&` (adresse-of) : pour obtenir l'adresse d'une variable
- Opérateur unaire `*` (pointeur d'indirection) : pour obtenir la variable pointée par un pointeur
- Opérateurs `->` (accès aux membres) et `[]` (accès à l'élément)
- Opérateurs arithmétiques `+` ,, `-` `++` et `--`
- Opérateurs de comparaison `==` ,,,, `!=` `<` `>` `<=` et `>=`

Pour plus d'informations sur les types de pointeurs, consultez Types pointeur.

> **NOTE**
>
> Toutes les opérations impliquant des pointeurs nécessitent un contexte unsafe. Le code qui contient des blocs unsafe doit être compilé avec l' `-unsafe` option de compilateur.

Opérateur d'adresse&

L'opérateur unaire `&` retourne l'adresse de son opérande :

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

L'opérande de l'opérateur `&` doit être une variable fixe. Les variables *fixes* se trouvent dans des emplacements de stockage qui ne sont pas affectés par le récupérateur de mémoire. Dans l'exemple précédent, la variable locale `number` est une variable fixe, car elle se trouve dans la pile. Les variables qui se trouvent dans des emplacements de stockage pouvant être affectés par le récupérateur de mémoire (par exemple, en étant déplacés) sont appelées variables *déplaçables*. Les champs d'objet et les éléments de tableau sont des exemples de variables déplaçables. Vous pouvez obtenir l'adresse d'une variable déplaçable si vous « corrigez », ou « épinglez », avec une `fixed` instruction. L'adresse obtenue est valide uniquement à l'intérieur du bloc d'une `fixed` instruction. L'exemple suivant montre comment utiliser une `fixed` instruction et l' `&` opérateur :

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

Vous ne pouvez pas obtenir l'adresse d'une constante ou d'une valeur.

Pour plus d'informations sur les variables fixes et déplaçables, consultez la section Variables fixes et déplaçables de la spécification du langage C#.

L'opérateur binaire `&` calcule la logique AND de ses opérandes booléens, ou la logique AND au niveau du bit de ses opérandes de type intégral.

Opérateur d'indirection de pointeur *

L'opérateur unaire d'indirection de pointeur `*` permet d'obtenir la variable vers laquelle pointe son opérande. Il est également appelé « opérateur de déréférence ». L'opérande de l'opérateur `*` doit être un type de pointeur.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

Vous ne pouvez pas appliquer l'opérateur `*` à une expression de type `void*`.

L'opérateur binaire `*` calcule le produit de ses opérandes numériques.

Opérateur d'accès aux membres de pointeur ->

L'opérateur `->` associe l'indirection de pointeur à l'accès aux membres. Autrement dit, si `x` est un pointeur de type `T*` et `y` est un membre accessible de type `T`, une expression de la forme

```
x->y
```

équivaut à :

```
(*x).y
```

L'exemple suivant illustre l'utilisation de l'opérateur `->` :

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

Vous ne pouvez pas appliquer l'opérateur `->` à une expression de type `void*`.

Opérateur d'accès aux éléments de pointeur []

Pour une expression `p` d'un type pointeur, l'accès à un élément de pointeur au format `p[n]` est évalué comme `*(p + n)`, où `n` doit être d'un type implicitement convertible en `int`, `uint`, `long` ou `ulong`. Pour plus d'informations sur le comportement de l'opérateur `+` avec les pointeurs, consultez la section Addition ou soustraction d'une valeur intégrale dans un pointeur.

L'exemple suivant montre comment accéder à des éléments tableau avec un pointeur et l'opérateur `[]` :

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Dans l'exemple précédent, une `stackalloc` expression alloue un bloc de mémoire sur la pile.

> **NOTE**
>
> L'opérateur d'accès aux éléments de pointeur ne recherche pas les erreurs de dépassement des limites.

Vous ne pouvez pas utiliser `[]` pour l'accès aux éléments de pointeur avec une expression de type `void*`.

Vous pouvez également utiliser l' `[]` opérateur pour l'accès à un élément ou un indexeur de tableau.

Opérateurs arithmétiques de pointeur

Vous pouvez effectuer les opérations arithmétiques suivantes avec des pointeurs :

- Ajouter ou soustraire une valeur intégrale dans un pointeur
- Soustraire deux pointeurs
- Incrémenter ou décrémenter un pointeur

Vous ne pouvez pas effectuer ces opérations avec des pointeurs de type `void*`.

Pour plus d'informations sur les opérations arithmétiques prises en charge avec les types numériques, consultez Opérateurs arithmétiques.

Ajout ou soustraction d'une valeur intégrale dans un pointeur

Pour un pointeur `p` de type `T*` et une expression `n` d'un type implicitement convertible en `int`, `uint`, `long` ou `ulong`, l'addition et la soustraction sont définies de la façon suivante :

- Les expressions `p + n` et `n + p` produisent un pointeur de type `T*` qui est obtenu en ajoutant `n * sizeof(T)` à l'adresse fournie par `p`.
- L'expression `p - n` produit un pointeur de type `T*` qui est obtenu en soustrayant `n * sizeof(T)` de l'adresse fournie par `p`.

L' `sizeof` opérateur obtient la taille d'un type en octets.

L'exemple suivant illustre l'utilisation de l'opérateur `+` avec un pointeur :

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Soustraction de pointeur

Pour deux pointeurs `p1` et `p2` de type `T*`, l'expression `p1 - p2` produit la différence entre les adresses fournies par `p1` et `p2`, divisée par `sizeof(T)`. Le type du résultat est `long`. C'est -à-dire que `p1 - p2` est calculé en tant que `((long)(p1) - (long)(p2)) / sizeof(T)`.

L'exemple suivant montre la soustraction d'un pointeur :

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Incrémenter et décrémenter des pointeurs

L'opérateur d'incrémentation `++` ajoute 1 à son opérande de pointeur. L'opérateur de décrémentation `--` soustrait 1 de son opérande de pointeur.

Les deux opérateurs sont pris en charge sous deux formes : suffixée (`p++` et `p--`) et préfixée (`++p` et `--p`). Le résultat de `p++` et `p--` correspond à la valeur de `p` *avant* l'opération. Le résultat de `++p` et `--p` correspond à la valeur de `p` *après* l'opération.

L'exemple suivant montre le comportement des opérateurs d'incrémentation suffixés et préfixés :

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Opérateurs de comparaison de pointeur

Vous pouvez utiliser les opérateurs `==` , `!=` , `<` , `>` , `<=` et `>=` pour comparer des opérandes de tout type de pointeur, y compris `void*` . Ces opérateurs comparent les adresses fournies par les deux opérandes comme s'il s'agissait d'entiers non signés.

Pour plus d'informations sur le comportement de ces opérateurs pour les opérandes d'autres types, consultez les articles Opérateurs d'égalité et Opérateurs de comparaison.

Précédence des opérateurs

La liste suivante présente les opérateurs relatifs aux pointeurs par ordre de précédence, de la plus élevée à la plus basse :

- Opérateurs suffixés d'incrémentation `x++` et de décrémentation `x--` , ainsi que les opérateurs `->` et `[]`
- Opérateurs préfixés d'incrémentation `++x` et de décrémentation `--x` , ainsi que les opérateurs `&` et `*`
- Opérateurs additifs `+` et `-`
- Opérateurs de comparaison `<` , `>` , `<=` et `>=`
- Opérateurs d'égalité `==` et `!=`

Utilisez des parenthèses (`()`) pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs.

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section priorité d'opérateur de l'article opérateurs c# .

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger les opérateurs de pointeur `&` , `*` , `->` et `[]` .

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Variables fixes et déplaçables
- address-of, opérateur

- Indirection de pointeur
- Accès aux membres de pointeur
- Accès aux éléments de pointeur
- Arithmétique des pointeurs
- Incrémentation et décrémentation des pointeurs
- Comparaison de pointeurs

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Types de pointeur
- mot clé unsafe
- mot clé Fixed
- stackalloc
- Opérateur sizeof

Opérateurs d'assignation (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

L'opérateur d'assignation `=` assigne la valeur de son opérande de droite à une variable, une propriété ou un élément d'indexeur donné par son opérande de gauche. Le résultat d'une expression d'assignation est la valeur assignée à l'opérande de gauche. L'opérande de droite doit être du même type que l'opérande de gauche, ou implicitement convertible vers le type de l'opérande de gauche.

L'opérateur `=` d'assignation est associatif à droite, autrement dit, une expression de la forme

```
a = b = c
```

est évaluée comme étant

```
a = (b = c)
```

L'exemple suivant illustre l'utilisation de l'opérateur d'assignation avec une variable locale, une propriété et un élément d'indexeur comme opérande de gauche :

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

Opérateur d'assignation ref

À partir de C# 7.3, vous pouvez utiliser l'opérateur d'assignation ref `= ref` pour réaffecter une variable locale ref ou une variable locale ref readonly. L'exemple suivant illustre l'utilisation de l'opérateur d'assignation ref :

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

Dans le cas de l'opérateur d'assignation de référence, les deux opérandes doivent être du même type.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```
x op= y
```

équivaut à :

```
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'assignation composée est prise en charge par les opérateurs arithmétiques, logiques booléens et logiques au niveau du bit et du décalage.

Assignation de fusion Null

À compter de C# 8,0, vous pouvez utiliser l'opérateur d'assignation de fusion Null `??=` pour assigner la valeur de son opérande droit à son opérande de gauche uniquement si l'opérande de gauche prend la valeur `null` . Pour plus d'informations, consultez les = l'article Operators .

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur d'assignation. Toutefois, il peut définir une conversion implicite vers un autre type. Ainsi, la valeur d'un type défini par l'utilisateur peut être assignée à une variable, une propriété ou un élément d'indexeur d'un autre type. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée. Toutefois, si un type défini par l'utilisateur surcharge un opérateur binaire `op` , l' `op=` opérateur, s'il existe, est également implicitement surchargé.

spécification du langage C#

Pour plus d'informations, voir la section Opérateurs d'assignation de la spécification du langage C#.

Pour plus d'informations sur l'opérateur d'assignation de référence `= ref` , consultez la Remarque relativeà la

proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- ref, mot clé

Expressions lambda (référence C#)

16/12/2020 • 21 minutes to read • Edit Online

Une *expression lambda* est une expression de l'une des deux formes suivantes :

- Expression lambda qui a une expression comme corps :

```
(input-parameters) => expression
```

- Instruction lambda qui a un bloc d'instructions comme corps :

```
(input-parameters) => { <sequence-of-statements> }
```

Utilisez l'opérateur de déclaration lambda `=>` pour séparer la liste des paramètres de l'expression lambda de son corps. Pour créer une expression lambda, vous spécifiez des paramètres d'entrée (le cas échéant) à gauche de l'opérateur lambda et une expression ou un bloc d'instructions de l'autre côté.

Toute expression lambda peut être convertie en type délégué. Le type délégué vers lequel une expression lambda peut être convertie est défini par les types de ses paramètres et de sa valeur de retour. Si une expression lambda ne retourne pas de valeur, elle peut être convertie en l'un des types délégués `Action` ; sinon, elle peut être convertie en l'un des types délégués `Func`. Par exemple, une expression lambda qui a deux paramètres et qui ne retourne aucune valeur peut être convertie en un délégué Action<T1,T2>. Une expression lambda qui a un paramètre et qui retourne une valeur peut être convertie en un délégué Func<T,TResult>. Dans l'exemple suivant, l'expression lambda `x => x * x` , qui spécifie un paramètre nommé `x` et retourne la valeur du `x` carré, est assignée à une variable d'un type délégué :

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Les expressions lambda d'expression peuvent également être converties en types d' arborescences d'expression , comme le montre l'exemple suivant :

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Vous pouvez utiliser des expressions lambda dans tout code qui requiert des instances de types délégués ou d'arborescences d'expressions, par exemple en tant qu'argument de la méthode Task.Run(Action) pour passer le code qui doit être exécuté en arrière-plan. Vous pouvez également utiliser des expressions lambda quand vous écrivez LINQ en C#, comme le montre l'exemple suivant :

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Lorsque la méthode Enumerable.Select est appelée avec une syntaxe fondée sur une méthode dans la classe System.Linq.Enumerable (par exemple, dans LINQ to Objects et LINQ to XML), le paramètre est un type délégué System.Func<T,TResult>. Quand vous appelez la Queryable.Select méthode dans la System.Linq.Queryable classe, par exemple dans LINQ to SQL, le type de paramètre est un type d'arborescence d'expression `Expression<Func<TSource,TResult>>` . Dans les deux cas, vous pouvez utiliser la même expression lambda pour spécifier la valeur de paramètre. Les deux appels `Select` semblent ainsi semblables alors qu'en fait le type des objets créés à partir des lambdas est différent.

Expressions lambdas

Une expression lambda comportant une expression à droite de l'opérateur `=>` est appelée *expression lambda*. Une expression lambda retourne le résultat de l'expression et prend la forme de base suivante :

```
(input-parameters) => expression
```

Le corps d'une expression lambda peut se composer d'un appel de méthode. Toutefois, si vous créez des arborescences d'expressions qui sont évaluées en dehors du contexte du Common Language Runtime (CLR) .net, par exemple dans SQL Server, vous ne devez pas utiliser les appels de méthode dans les expressions lambda. Les méthodes n'auront aucune signification en dehors du contexte du Common Language Runtime (CLR) .NET.

Instructions lambda

Une instruction lambda ressemble à une expression lambda, sauf que ses instructions sont placées entre accolades :

```
(input-parameters) => { <sequence-of-statements> }
```

Le corps d'une instruction lambda peut se composer d'un nombre illimité d'instructions ; toutefois, en pratique, leur nombre est généralement de deux ou trois.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Vous ne pouvez pas utiliser des lambdas d'instruction pour créer des arborescences d'expressions.

Paramètres d'entrée d'une expression lambda

Placez les paramètres d'entrée d'une expression lambda entre parenthèses. Spécifiez des

paramètres d'entrée de zéro avec des parenthèses vides :

```
Action line = () => Console.WriteLine();
```

Si une expression lambda a un seul paramètre d'entrée, les parenthèses sont facultatives :

```
Func<double, double> cube = x => x * x * x;
```

Deux ou plusieurs paramètres d'entrée sont séparés par des virgules :

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Parfois, le compilateur ne peut pas déduire les types de paramètres d'entrée. Vous pouvez spécifier les types explicitement comme dans l'exemple suivant :

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Les types de paramètres d'entrée doivent être tous explicites ou tous implicites ; sinon, une erreur de compilateur CS0748 se produit.

À compter de C# 9,0, vous pouvez utiliser des éléments ignorés pour spécifier au moins deux paramètres d'entrée d'une expression lambda qui ne sont pas utilisés dans l'expression :

```
Func<int, int, int> constant = (_, _) => 42;
```

Les paramètres d'abandon lambda peuvent être utiles lorsque vous utilisez une expression lambda pour fournir un gestionnaire d'événements.

> **NOTE**
> À des fins de compatibilité descendante, si un seul paramètre d'entrée est nommé `_`, alors, dans une expression lambda, `_` est traité comme le nom de ce paramètre.

Lambdas asynchrones

Vous pouvez facilement créer des expressions et des instructions lambda qui incorporent le traitement asynchrone à l'aide des mots clés async et await . Par exemple, l'exemple Windows Forms suivant contient un gestionnaire d'événements qui appelle et attend une méthode async `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Vous pouvez ajouter le même gestionnaire d'événements en utilisant une expression lambda asynchrone. Pour ajouter ce gestionnaire, ajoutez un modificateur `async` avant la liste des paramètres lambda, comme dans l'exemple suivant :

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Pour plus d'informations sur la création et l'utilisation des méthodes Async, consultez programmation asynchrone avec Async et await.

Expressions lambda et tuples

À compter de C# 7,0, le langage C# offre une prise en charge intégrée des tuples. Vous pouvez fournir un tuple comme argument à une expression lambda, et votre expression lambda peut aussi retourner un tuple. Dans certains cas, le compilateur C# utilise l'inférence de type pour déterminer les types des composants du tuple.

Vous définissez un tuple en plaçant entre des parenthèses une liste de ses composants avec des virgules comme séparateur. L'exemple suivant utilise un tuple à trois composants pour passer une séquence de nombres à une expression lambda, qui double chaque valeur et retourne un tuple à trois composants contenant le résultat des multiplications.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 *
ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

En règle générale, les champs d'un tuple sont nommés `Item1`, `Item2`, etc. Toutefois, vous pouvez définir un tuple avec des composants nommés, comme le montre l'exemple suivant.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 *
ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Pour plus d'informations sur les tuples C#, consultez types de tuples.

Lambdas avec les opérateurs de requête standard

LINQ to Objects, parmi d'autres implémentations, a un paramètre d'entrée dont le type fait partie de la famille Func<TResult> de délégués génériques. Ces délégués utilisent des paramètres de type pour définir le nombre et le type des paramètres d'entrée, ainsi que le type de retour du délégué. Les délégués `Func` sont très utiles pour l'encapsulation des expressions définies par l'utilisateur appliquées à chaque élément dans un jeu de données sources. Prenons par exemple le type délégué Func<T,TResult> :

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Ce délégué peut être instancié comme une instance `Func<int, bool>`, où `int` est un paramètre d'entrée et `bool` la valeur de retour. La valeur de retour est toujours spécifiée dans le dernier paramètre de type. Par exemple, `Func<int, string, bool>` définit un délégué avec deux paramètres d'entrée, `int` et `string`, et un type de retour `bool`. Le délégué `Func` suivant, lorsqu'il est appelé, retourne une valeur booléenne indiquant si le paramètre d'entrée est égal à cinq :

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

Vous pouvez aussi fournir une expression lambda quand le type d'argument est Expression<TDelegate>, par exemple, dans les opérateurs de requête standard définis dans le type Queryable. Quand vous spécifiez un argument Expression<TDelegate>, l'expression lambda est compilée en arborescence de l'expression.

L'exemple suivant utilise l'opérateur de requête standard Count :

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

Le compilateur peut déduire le type du paramètre d'entrée, ou vous pouvez également le spécifier explicitement. Cette expression lambda particulière compte les entiers (`n`) qui, lorsqu'ils sont

divisés par deux, ont un reste de 1.

L'exemple suivant produit une séquence qui contient tous les éléments du tableau `numbers` précédant le 9, car c'est le premier nombre de la séquence qui ne remplit pas la condition :

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

L'exemple suivant spécifie plusieurs paramètres d'entrée en les plaçant entre parenthèses. La méthode retourne tous les éléments du tableau `numbers` jusqu'à ce qu'elle rencontre un nombre dont la valeur est inférieure à sa position ordinale dans le tableau :

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Inférence de type et expressions lambda

Il n'est généralement pas nécessaire, avec les expressions lambda, de spécifier un type pour les paramètres d'entrée, car le compilateur peut le déduire du corps de l'expression, des types de paramètres et d'autres facteurs, comme le décrit la spécification du langage C#. Pour la plupart des opérateurs de requête standard, la première entrée est le type des éléments dans la séquence source. Si vous interrogez un `IEnumerable<Customer>`, le type de la variable d'entrée est inféré comme étant un objet `Customer`, ce qui signifie que vous avez accès à ses méthodes et à ses propriétés :

```
customers.Where(c => c.City == "London");
```

Voici les règles générales de l'inférence de type pour les expressions lambda :

- Le lambda doit contenir le même nombre de paramètres que le type délégué.
- Chaque paramètre d'entrée dans le lambda doit être implicitement convertible en son paramètre de délégué correspondant.
- La valeur de retour du lambda (le cas échéant) doit être implicitement convertible en type de retour du délégué.

Notez que les expressions lambda n'ont pas de type en elles-mêmes, car le système de type commun (CTS, Common Type System) ne comporte aucun concept intrinsèque « d'expression lambda ». Toutefois, il est parfois commode de parler de manière informelle du « type » d'une expression lambda. Dans ce cas, le type fait référence au type délégué ou au type Expression dans lequel est convertie l'expression lambda.

Capture des variables externes et de l'étendue variable dans les expressions lambda

Les expressions lambda peuvent faire référence à des *variables externes*. Il s'agit de variables dans l'étendue de la méthode qui définit l'expression lambda, ou dans l'étendue du type qui contient

l'expression lambda. Les variables capturées de cette manière sont stockées pour une utilisation dans l'expression lambda, même si les variables se trouvent en dehors de la portée et sont récupérées par le garbage collector. Une variable externe doit être assignée de manière précise pour pouvoir être utilisée dans une expression lambda. L'exemple suivant illustre ces règles :

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable:
{equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

Les règles suivantes s'appliquent à la portée des variables dans les expressions lambda :

- Une variable qui est capturée ne sera pas récupérée par le garbage collector avant que le délégué qui le référence soit disponible pour le garbage collection.
- Les variables introduites dans une expression lambda ne sont pas visibles dans la méthode englobante.

- Une expression lambda ne peut pas capturer directement un paramètre in, ref ou out dans la méthode englobante.
- Une instruction return dans une expression lambda ne provoque pas le retour de la méthode englobante.
- Une expression lambda ne peut pas contenir d'instruction goto, break ou continue si la cible de cette instruction de saut se trouve en dehors du bloc de l'expression lambda. Il est également incorrect d'avoir une instruction de saut en dehors du bloc de l'expression lambda si la cible se trouve à l'intérieur du bloc.

À compter de C# 9,0, vous pouvez appliquer le `static` modificateur à une expression lambda pour empêcher la capture non intentionnelle de variables locales ou l'état de l'instance par le lambda :

```
Func<double, double> square = static x => x * x;
```

Une expression lambda statique ne peut pas capturer les variables locales ou l'état de l'instance à partir des étendues englobantes, mais peut référencer des membres statiques et des définitions de constantes.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Pour plus d'informations sur les fonctionnalités ajoutées dans C# 9,0, consultez les notes de proposition de fonctionnalités suivantes :

- Paramètres d'abandon lambda
- Fonctions anonymes statiques

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- LINQ (Language Integrated Query)
- Arborescences d'expressions
- Fonctions locales et expressions lambda
- Exemples Visual Studio 2008 C# (voir les fichiers d'exemples de requêtes LINQ et le programme XQuery)

+ et +=, opérateurs (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Les `+` `+=` opérateurs et sont pris en charge par les types numériques intégraux et à virgule flottante intégrés, le type de chaîne et les types délégués .

Pour plus d'informations sur l'opérateur arithmétique `+`, consultez les sections Opérateurs plus et moins unaires et Opérateur d'addition + de l'article Opérateurs arithmétiques.

Concaténation de chaînes

Quand l'un des opérandes ou les deux sont de type chaîne, l' `+` opérateur concatène les représentations sous forme de chaîne de ses opérandes (la représentation sous forme de chaîne de `null` est une chaîne vide) :

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

À compter de C# 6, l' interpolation de chaîne offre un moyen plus pratique de mettre en forme les chaînes :

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Combinaison de délégués

Pour les opérandes du même type délégué, l'opérateur `+` retourne une nouvelle instance de délégué qui, lorsqu'elle est appelée, appelle l'opérande de partie gauche, puis l'opérande de partie droite. Si un des opérandes est `null`, l'opérateur `+` retourne la valeur de l'autre opérande (qui peut également être `null`). L'exemple suivant montre comment les délégués peuvent être combinés avec l'opérateur `+` :

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

Pour effectuer la suppression d'un délégué, utilisez l' `-` opérateur.

Pour plus d'informations sur les types de délégués, consultez Délégués.

Opérateur d'assignation d'addition +=

Expression utilisant l'opérateur `+=`, par exemple

```
x += y
```

équivaut à :

```
x = x + y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'opérateur `+=` :

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer();  // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer();  // output: ab
```

Vous utilisez également l'opérateur `+=` pour spécifier une méthode de gestionnaire d'événements lorsque vous vous abonnez à un événement. Pour plus d'informations, consultez Guide pratique pour s'abonner et annuler l'abonnement à des événements.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger l'opérateur `+`. Quand un opérateur binaire `+` est surchargé, l'opérateur `+=` est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement l'opérateur `+=`.

spécification du langage C#

Pour plus d'informations, consultez les sections Opérateur unaire plus et Opérateur d'addition de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Concaténation de plusieurs chaînes
- Événements
- Opérateurs arithmétiques
- opérateurs-and-=

Opérateurs - et -=, opérateurs - (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Les `-` `-=` opérateurs et sont pris en charge par les types numériques intégraux et à virgule flottante intégrés et les types délégués .

Pour plus d'informations sur l'opérateur arithmétique `-`, consultez les sections Opérateurs plus et moins unaires et Opérateur de soustraction - de l'article Opérateurs arithmétiques.

Suppression de délégué

Pour les opérandes du même type délégué, l'opérateur `-` retourne une instance de délégué qui est calculée comme suit :

- Si les deux opérandes ont des valeurs non Null et que la liste d'appel de l'opérande de partie droite est une sous-liste contiguë correcte de la liste d'appel de l'opérande de partie droite, le résultat de l'opération est une nouvelle liste d'appel obtenue en supprimant les entrées de l'opérande de partie droite à partir de la liste d'appel de l'opérande de gauche. Si la liste des opérandes de partie droite correspond à plusieurs sous-listes contiguës dans la liste des opérandes de partie gauche, seule la sous-liste correspondante la plus à droite est supprimée. Si la suppression aboutit à une liste vide, le résultat est `null`.

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  abbaab();  // output: abbaab
  Console.WriteLine();

  var ab = a + b;
  var abba = abbaab - ab;
  abba();  // output: abba
  Console.WriteLine();

  var nihil = abbaab - abbaab;
  Console.WriteLine(nihil is null);  // output: True
  ```

- Si la liste d'appel de l'opérande de partie droite n'est pas une sous-liste contiguë correcte de la liste d'appel de l'opérande de partie gauche, le résultat de l'opération est l'opérande de partie gauche. Par exemple, la suppression d'un délégué qui ne fait pas partie du délégué multicast ne fait rien et génère un délégué multicast inchangé.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first();  // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output: True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed();  // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged();  // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output: True
```

L'exemple précédent montre également que durant la suppression de délégué, les instances de délégués sont comparées. Par exemple, les délégués qui sont produits à partir de l'évaluation d'expressions lambda identiques ne sont pas égaux. Pour plus d'informations sur l'égalité des délégués, consultez la section Opérateurs d'égalité de délégués de la spécification du langage C#.

- Si l'opérande de partie gauche est `null`, le résultat de l'opération est `null`. Si l'opérande de partie droite est `null`, le résultat de l'opération est l'opérande de partie gauche.

```
Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null);  // output: True

var first = a - null;
a();  // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

Pour combiner des délégués, utilisez l' `+` opérateur.

Pour plus d'informations sur les types de délégués, consultez Délégués.

Opérateur d'assignation de soustraction -=

Expression utilisant l'opérateur `-=`, par exemple

```
x -= y
```

équivaut à :

```
x = x - y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'opérateur `-=` :

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

Vous utilisez également l'opérateur `-=` pour spécifier une méthode de gestionnaire d'événements à supprimer quand vous vous désabonnez d'un événement. Pour plus d'informations, consultez Comment s'abonner et annuler l'abonnement à des événements.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger l'opérateur `-`. Quand un opérateur binaire `-` est surchargé, l'opérateur `-=` est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement l'opérateur `-=`.

spécification du langage C#

Pour plus d'informations, consultez les sections Opérateur moins unaire et Opérateur de soustraction de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Événements
- Opérateurs arithmétiques
- opérateurs + et + =

Opérateur ?: (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

L'opérateur conditionnel `?:` , également appelé opérateur conditionnel ternaire, évalue une expression booléenne et retourne le résultat de l'une des deux expressions, selon que l'expression booléenne a pour valeur `true` ou `false` .

Voici la syntaxe de l'opérateur conditionnel :

```
condition ? consequent : alternative
```

L'expression `condition` doit donner `true` ou `false`. Si `condition` prend la valeur `true`, l'expression `consequent` est évaluée et son résultat devient le résultat de l'opération. Si `condition` prend la valeur `false`, l'expression `alternative` est évaluée et son résultat devient le résultat de l'opération. Soit `consequent`, soit `alternative` est évaluée.

À compter de C# 9,0, les expressions conditionnelles sont de type cible. Autrement dit, si un type cible d'une expression conditionnelle est connu, les types de `consequent` et `alternative` doivent être implicitement convertibles en type cible, comme le montre l'exemple suivant :

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

Si le type cible d'une expression conditionnelle est inconnu (par exemple, lorsque vous utilisez le `var` mot clé) ou en C# 8,0 et les versions antérieures, le type de `consequent` et `alternative` doit être identique ou il doit y avoir une conversion implicite d'un type à l'autre :

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

L'opérateur conditionnel est associatif à droite ; autrement dit, une expression de la forme :

```
a ? b : c ? d : e
```

est évaluée comme étant

```
a ? b : (c ? d : e)
```

TIP

Vous pouvez utiliser l'appareil mnémonique suivant pour vous souvenir du mode d'évaluation de l'opérateur conditionnel :

```
is this condition true ? yes : no
```

L'exemple suivant illustre l'utilisation de l'opérateur conditionnel :

```
double sinc(double x) => x != 0.0 ? Math.Sin(x) / x : 1;

Console.WriteLine(sinc(0.1));
Console.WriteLine(sinc(0.0));
// Output:
// 0.998334166468282
// 1
```

Expression ref conditionnelle

À compter de C# 7,2, une variable locale Ref locale ou ref ReadOnly peut être assignée de manière conditionnelle avec une expression Ref conditionnelle. Vous pouvez également utiliser une expression Ref conditionnelle comme valeur de retour de référence ou comme `ref` argument de méthode.

La syntaxe d'une expression Ref conditionnelle est la suivante :

```
condition ? ref consequent : ref alternative
```

À l'instar de l'opérateur conditionnel d'origine, une expression Ref conditionnelle évalue uniquement l'une des deux expressions : `consequent` ou `alternative` .

Dans le cas d'une expression Ref conditionnelle, le type de `consequent` et `alternative` doit être identique. Les expressions Ref conditionnelles ne sont pas de type cible.

L'exemple suivant illustre l'utilisation d'une expression Ref conditionnelle :

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Opérateur conditionnel et instruction `if..else`

L'utilisation de l'opérateur conditionnel au lieu d'une instruction if-else peut entraîner un code plus concis dans les cas où vous avez besoin de calculer une valeur de manière conditionnelle. L'exemple suivant montre

deux façons de classer un entier comme négatif ou non :

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur conditionnel.

spécification du langage C#

Pour plus d'informations, voir la section Opérateur conditionnel de la spécification du langage C#.

Pour plus d'informations sur les fonctionnalités ajoutées dans C# 7,2 et versions ultérieures, consultez les notes de proposition de fonctionnalités suivantes :

- Expressions Ref conditionnelles (C# 7,2)
- Expression conditionnelle typée cible (C# 9,0)

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- if-else, instruction
- Opérateurs ?. et ?[]
- ?? et ?? =, opérateurs
- ref, mot clé

! (null-indulgent avec), opérateur (référence C#)

16/12/2020 • 4 minutes to read • Edit Online

Disponible en C# 8,0 et versions ultérieures, l'opérateur postfix unaire `!` est l'opérateur null-indulgent avec, ou suppression de valeur null. Dans un contexte d'annotation Nullableactivé, vous utilisez l'opérateur null-indulgent avec pour déclarer que l'expression `x` d'un type référence n'est pas `null` : `x!` . L'opérateur préfixé unaire `!` est l' opérateur de négation logique.

L'opérateur null-indulgent avec n'a aucun effet au moment de l'exécution. Elle affecte uniquement l'analyse du déroulement statique du compilateur en modifiant l'État null de l'expression. Au moment de l'exécution, l'expression `x!` prend la valeur du résultat de l'expression sous-jacente `x` .

Pour plus d'informations sur la fonctionnalité des types de référence Nullable, consultez types de référence Nullable.

Exemples

L'un des cas d'usage de l'opérateur null-indulgent avec consiste à tester la logique de validation d'argument. Par exemple, considérons la classe suivante :

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

À l'aide de l' infrastructure de test MSTest, vous pouvez créer le test suivant pour la logique de validation dans le constructeur :

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Sans l'opérateur null-indulgent avec, le compilateur génère l'avertissement suivant pour le code précédent : `Warning CS8625: Cannot convert null literal to non-nullable reference type` . En utilisant l'opérateur null-indulgent avec, vous informez le compilateur que le passage `null` est attendu et ne doit pas être averti de.

Vous pouvez également utiliser l'opérateur null-indulgent avec lorsque vous savez absolument qu'une expression ne peut pas être, `null` mais que le compilateur ne gère pas. Dans l'exemple suivant, si la `IsValid` méthode retourne `true` , son argument n'est pas `null` et vous pouvez la déréférencer en toute sécurité :

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}
```

Sans l'opérateur null-indulgent avec, le compilateur génère l'avertissement suivant pour le `p.Name` Code : `Warning CS8602: Dereference of a possibly null reference` .

Si vous pouvez modifier la `IsValid` méthode, vous pouvez utiliser l'attribut NotNullWhen pour informer le compilateur qu'un argument de la `IsValid` méthode ne peut pas être `null` lorsque la méthode retourne `true` :

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}
```

Dans l'exemple précédent, vous n'avez pas besoin d'utiliser l'opérateur null-indulgent avec, car le compilateur dispose de suffisamment d'informations pour déterminer qu'il `p` ne peut pas se trouver `null` à l'intérieur de l' `if` instruction. Pour plus d'informations sur les attributs qui vous permettent de fournir des informations supplémentaires sur l'État null d'une variable, consultez mettre à niveau les API avec des attributs pour définir des attentes null.

spécification du langage C#

Pour plus d'informations, consultez la section opérateur null-indulgent avec du brouillon de la spécification des types de référence Nullable.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Didacticiel : concevoir avec des types de référence Nullable

?? et ?? =, opérateurs (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

L'opérateur de fusion null `??` retourne la valeur de l'opérande de gauche si elle n'est pas `null` ; sinon, il évalue l'opérande de droite et retourne son résultat. L'opérateur `??` n'évalue pas son opérande droit si l'opérande gauche a la valeur non null.

Disponible en C# 8,0 et versions ultérieures, l'opérateur d'assignation de fusion Null `??=` affecte la valeur de son opérande droit à son opérande de gauche uniquement si l'opérande de gauche est évalué à `null` . L'opérateur `??=` n'évalue pas son opérande droit si l'opérande gauche a la valeur non null.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

L'opérande gauche de l' `??=` opérateur doit être une variable, une propriétéou un élément d' indexeur .

En C# 7,3 et versions antérieures, le type de l'opérande gauche de l' `??` opérateur doit être un type référence ou un type valeur Nullable. À compter de C# 8,0, cette spécification est remplacée par ce qui suit : le type de l'opérande gauche des `??` opérateurs et `??=` ne peut pas être un type valeur non Nullable. En particulier, à compter de C# 8,0, vous pouvez utiliser les opérateurs de fusion Null avec des paramètres de type sans contrainte :

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

Les opérateurs de fusion Null sont associatifs à droite. Autrement dit, les expressions de la forme

```
a ?? b ?? c
d ??= e ??= f
```

sont évaluées comme

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Exemples

Les `??` `??=` opérateurs et peuvent être utiles dans les scénarios suivants :

- Dans les expressions avec les opérateurs conditionnels null ?. et ? [], vous pouvez utiliser l' `??` opérateur pour fournir une autre expression à évaluer au cas où le résultat de l'expression avec des

opérations conditionnelles null est `null` :

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum);  // output: NaN
```

- Quand vous utilisez des types valeur Nullable et que vous devez fournir une valeur d'un type valeur sous-jacent, utilisez l' `??` opérateur pour spécifier la valeur à fournir dans le cas où une valeur de type Nullable est `null` :

  ```
  int? a = null;
  int b = a ?? -1;
  Console.WriteLine(b);  // output: -1
  ```

  Utilisez la méthode Nullable<T>.GetValueOrDefault() si la valeur à utiliser quand une valeur de type Nullable est `null` doit être la valeur par défaut du type valeur sous-jacent.

- À compter de C# 7,0, vous pouvez utiliser une `throw` expression comme opérande de droite de l' `??` opérateur pour rendre le code de vérification d'argument plus concis :

  ```
  public string Name
  {
      get => name;
      set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
  }
  ```

  L'exemple précédent montre également comment utiliser des membres expression-bodied pour définir une propriété.

- À compter de C# 8,0, vous pouvez utiliser l' `??=` opérateur pour remplacer le code du formulaire.

  ```
  if (variable is null)
  {
      variable = expression;
  }
  ```

  par le code suivant :

  ```
  variable ??= expression;
  ```

Capacité de surcharge de l'opérateur

Les opérateurs `??` et `??=` ne peuvent pas être surchargés.

spécification du langage C#

Pour plus d'informations sur l' `??` opérateur, consultez la section opérateur de fusion Null de la spécification du langage C#.

Pour plus d'informations sur l' `??=` opérateur, consultez la Remarque relativeà la proposition de

fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs ?. et ?[]
- ?:, opérateur

Opérateur >= (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Le `=>` jeton est pris en charge sous deux formes : comme opérateur lambda et comme séparateur d'un nom de membre et l'implémentation de membre dans une définition de corps d'expression.

Opérateur lamdba

Dans les expressions lambda, l'opérateur lambda `=>` sépare les paramètres d'entrée sur le côté gauche du corps lambda du côté droit.

L'exemple suivant utilise la fonctionnalité LINQ avec la syntaxe de méthode pour illustrer l'utilisation d'expressions lambda :

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Les paramètres d'entrée d'une expression lambda sont fortement typés au moment de la compilation. Lorsque le compilateur peut déduire les types de paramètres d'entrée, comme dans l'exemple précédent, vous pouvez omettre des déclarations de type. Si vous devez spécifier le type des paramètres d'entrée, vous devez le faire pour chaque paramètre, comme le montre l'exemple suivant :

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);   // output: 280
```

L'exemple suivant montre comment définir une expression lambda sans paramètres d'entrée :

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Pour plus d'informations, consultez Expressions lambda.

Définition de corps d'expression

La syntaxe générale d'une définition de corps d'expression est la suivante :

```
member => expression;
```

où `expression` est une expression valide. Le type de retour de `expression` doit être implicitement convertible en type de retour du membre. Si le type de retour du membre est `void` ou si le membre est un constructeur, un finaliseur ou un accesseur de propriété ou d'indexeur `set`, `expression` doit être une *expression d'instruction*. Étant donné que le résultat de l'expression est ignoré, le type de retour de cette expression peut être n'importe

quel type.

L'exemple suivant montre une définition de corps d'expression pour une méthode `Person.ToString` :

```
public override string ToString() => $"{fname} {lname}".Trim();
```

Il s'agit d'une version raccourcie de la définition de méthode suivante :

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

Les définitions de corps d'expression pour les méthodes, les opérateurs et les propriétés en lecture seule sont prises en charge à partir de C# 6. Les définitions de corps d'expression pour les constructeurs, les finaliseurs et les accesseurs de propriété et d'indexeur sont pris en charge à compter de C# 7,0.

Pour plus d'informations, consultez Membres expression-bodied.

Capacité de surcharge de l'opérateur

L'opérateur `=>` ne peut pas être surchargé.

spécification du langage C#

Pour plus d'informations sur l'opérateur lambda, consultez la section expressions de fonction anonymes de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

Opérateur :: (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Utilisez le qualificateur d'alias `::` d'espace de noms pour accéder à un membre d'un espace de noms avec alias. Vous pouvez utiliser le `::` qualificateur uniquement entre deux identificateurs. L'identificateur de gauche peut être un des alias suivants :

- Alias d'espace de noms créé avec une directive using alias:

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
point.Y);
}
```

- Un alias externe.

- L'alias `global`, qui est l'alias d'espace de noms global. L'espace de noms global est l'espace de noms qui contient des espaces de noms et des types qui ne sont pas déclarés dans un espace de noms nommé. Quand il est utilisé avec le qualificateur `::`, l'alias `global` référence toujours l'espace de noms global, même si l'alias d'espace de noms `global` défini par l'utilisateur est présent.

  L'exemple suivant utilise l'alias `global` pour accéder à l'espace de noms .NET System, qui est un membre de l'espace de noms global. Sans l'alias `global`, l'accès se ferait à l'espace de noms `System` défini par l'utilisateur, qui est un membre de l'espace de noms `MyCompany.MyProduct` :

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

> **NOTE**
>
> Le mot clé `global` est l'alias d'espace de noms global seulement quand il s'agit de l'identificateur de gauche du qualificateur `::`.

Vous pouvez également utiliser le `.` jeton pour accéder à un membre d'un espace de noms avec alias. Toutefois, le `.` jeton est également utilisé pour accéder à un membre de type. Le qualificateur `::` garantit que son identificateur de gauche référence toujours un alias d'espace de noms, même s'il existe un type ou un espace de noms portant le même nom.

spécification du langage C#

Pour plus d'informations, consultez la section Qualificateurs d'alias d'espace de noms de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Using Namespaces

Opérateur await (informations de référence sur C#)

02/11/2020 • 7 minutes to read • Edit Online

L'opérateur `await` suspend l'évaluation de la méthode async englobante jusqu'à ce que l'opération asynchrone représentée par son opérande se termine. Une fois l'opération asynchrone terminée, l'opérateur `await` retourne le résultat de l'opération, le cas échéant. Lorsque l' `await` opérateur est appliqué à l'opérande qui représente une opération déjà terminée, il retourne le résultat de l'opération immédiatement sans suspension de la méthode englobante. L'opérateur `await` ne bloque pas le thread qui évalue la méthode async. Lorsque l' `await` opérateur interrompt la méthode Async englobante, le contrôle retourne à l'appelant de la méthode.

Dans l'exemple suivant, la méthode HttpClient.GetByteArrayAsync retourne l'instance `Task<byte[]>`, qui représente une opération asynchrone qui produit un tableau d'octets au moment où elle se termine. Tant que l'opération n'est pas terminée, l'opérateur `await` suspend la méthode `DownloadDocsMainPageAsync`. Quand la méthode `DownloadDocsMainPageAsync` est suspendue, le contrôle est retourné à la méthode `Main`, qui est l'appelant de `DownloadDocsMainPageAsync`. La méthode `Main` s'exécute jusqu'à ce qu'elle ait besoin du résultat de l'opération asynchrone effectuée par la méthode `DownloadDocsMainPageAsync`. Une fois que tous les octets ont été obtenus par GetByteArrayAsync, le reste de la méthode `DownloadDocsMainPageAsync` est évalué. Après cela, le reste de la méthode `Main` est évalué.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

L'exemple précédent utilise la `Main` méthode Async, qui est possible à partir de C# 7,1. Pour plus

d'informations, consultez la section Opérateur await dans la méthode Main.

> **NOTE**
> Pour obtenir une introduction à la programmation asynchrone, consultez Programmation asynchrone avec async et await. La programmation asynchrone avec `async` et `await` suit le modèle asynchrone basé sur les tâches.

Vous pouvez utiliser l'opérateur `await` uniquement dans une méthode, une expression lambda ou une méthode anonyme modifiée par le mot clé async. Dans une méthode Async, vous ne pouvez pas utiliser l' `await` opérateur dans le corps d'une fonction synchrone, à l'intérieur du bloc d'une instruction locket dans un contexte unsafe .

L'opérande de l'opérateur `await` est généralement un des types .NET suivants : Task, Task<TResult>, ValueTask ou ValueTask<TResult>. Cependant, n'importe quelle expression awaitable peut être l'opérande de l'opérateur `await`. Pour plus d'informations, consultez la section Expressions awaitable de la spécification du langage C#.

Le type d'expression `await t` est `TResult` si le type d'expression `t` est Task<TResult> ou ValueTask<TResult>. Si le type de `t` est Task ou ValueTask, le type de `await t` est `void`. Dans les deux cas, si `t` lève une exception, `await t` lève à nouveau l'exception. Pour plus d'informations sur le gestion des exceptions, consultez la section Exceptions dans les méthodes async de l'article Instruction try-catch.

Les `async` `await` Mots clés et sont disponibles dans C# 5 et versions ultérieures.

Flux asynchrones et supprimables

À compter de C# 8,0, vous pouvez utiliser des flux asynchrones et des sources jetables.

Vous utilisez l' `await foreach` instruction pour consommer un flux de données asynchrone. Pour plus d'informations, consultez l'article sur les `foreach` instructions et la section relative aux flux asynchrones de l'article Nouveautés de C# 8,0 .

Vous utilisez l' `await using` instruction pour travailler avec un objet pouvant être supprimé de manière asynchrone, autrement dit, un objet d'un type qui implémente une IAsyncDisposable interface. Pour plus d'informations, consultez la section utilisation de Async jetable de l'article implémentation d'une méthode DisposeAsync .

Opérateur await dans la méthode Main

À compter de C# 7,1, la `Main` méthode, qui est le point d'entrée de l'application, peut retourner `Task` ou, ce qui lui permet `Task<int>` d'être asynchrone afin que vous puissiez utiliser l' `await` opérateur dans son corps. Dans les versions antérieures de C#, pour être certain que la méthode `Main` attend la fin d'une opération asynchrone, vous pouvez récupérer la valeur de la propriété Task<TResult>.Result de l'instance Task<TResult> retournée par la méthode asyn correspondante. Pour les opérations asynchrones qui ne produisent pas de valeur, vous pouvez appeler la méthode Task.Wait. Pour plus d'informations sur la façon de sélectionner la version linguistique, consultez contrôle de version du langage C#.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions await de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#

- Opérateurs et expressions C#
- async
- Modèle de programmation asynchrone des tâches
- Programmation asynchrone
- Async en détail
- Procédure pas à pas : accès au Web avec Async et await
- Didacticiel : générer et utiliser des flux asynchrones à l'aide de C# 8,0 et .NET Core 3,0

expressions de valeur par défaut (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Une expression de valeur par défaut produit la valeur par défaut d'un type. Il existe deux types d'expressions de valeur par défaut : l'appel d' opérateur par défaut et un littéral par défaut.

Vous utilisez également le `default` mot clé comme étiquette de cas par défaut dans une `switch` instruction.

Opérateur default

L'argument de l'opérateur `default` doit avoir le nom d'un type ou d'un paramètre de type, comme le montre l'exemple suivant :

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

littéral par défaut

À partir de C# 7.1, vous pouvez utiliser le littéral `default` pour produire la valeur par défaut d'un type lorsque le compilateur peut déduire le type d'expression. L'expression littérale `default` génère la même valeur que l'expression `default(T)`, où `T` est le type déduit. Vous pouvez utiliser le littéral `default` dans les cas suivants :

- Dans l'affectation ou l'initialisation d'une variable.
- Dans la déclaration de la valeur par défaut d'un paramètre de méthode facultatif.
- Dans un appel de méthode pour fournir une valeur d'argument.
- Dans une `return` instruction ou en tant qu'expression dans un membre d'expression.

L'exemple suivant illustre l'utilisation du littéral `default` :

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0) ]
```

spécification du langage C#

Pour plus d'informations, voir la section Expressions de valeur par défaut de la spécification du langage C#.

Pour plus d'informations sur le littéral `default`, voir la proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Valeurs par défaut des types C#
- Génériques en .NET

opérateur délégué (Référence C#)

02/11/2020 • 3 minutes to read • Edit Online

L'opérateur `delegate` crée une méthode anonyme qui peut être convertie en un type délégué :

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

NOTE

À partir C# 3, les expressions lambda offrent un moyen plus concis et plus expressif de créer une fonction anonyme. Utilisez `=>` opérateur pour construire une expression lambda :

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

Pour plus d'informations sur les fonctionnalités des expressions lambda, par exemple la capture de variables externes, consultez Expressions lambda.

Lorsque vous utilisez l'opérateur `delegate`, vous pouvez omettre la liste de paramètres. Dans ce cas, la méthode anonyme créée peut être convertie en un type délégué avec n'importe quelle liste de paramètres, comme le montre l'exemple suivant :

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

C'est la seule fonctionnalité des méthodes anonymes qui n'est pas prise en charge par les expressions lambda. Dans tous les autres cas, une expression lambda est le moyen préféré pour écrire du code incorporé.

À compter de C# 9,0, vous pouvez utiliser des éléments ignorés pour spécifier au moins deux paramètres d'entrée d'une méthode anonyme qui ne sont pas utilisés par la méthode :

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4));  // output: 42
```

À des fins de compatibilité descendante, si un seul paramètre est nommé `_`, `_` est traité comme le nom de ce paramètre dans une méthode anonyme.

À compter de C# 9,0, vous pouvez également utiliser le `static` modificateur au niveau de la déclaration d'une méthode anonyme :

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4));  // output: 14
```

Une méthode anonyme statique ne peut pas capturer les variables locales ou l'état de l'instance à partir des étendues englobantes.

Vous utilisez également le mot clé `delegate` pour déclarer un type délégué.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- => opérateur

nameof, expression (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Une `nameof` expression produit le nom d'une variable, d'un type ou d'un membre en tant que constante de chaîne :

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

Comme le montre l'exemple précédent, dans le cas d'un type et d'un espace de noms, le nom produit n'est généralement pas complet.

Dans le cas des identificateurs textuels, le `@` caractère n'est pas la partie d'un nom, comme le montre l'exemple suivant :

```
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

Une `nameof` expression est évaluée au moment de la compilation et n'a aucun effet au moment de l'exécution.

Vous pouvez utiliser une `nameof` expression pour rendre le code de vérification des arguments plus gérable :

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

Une `nameof` expression est disponible en C# 6 et versions ultérieures.

spécification du langage C#

Pour plus d'informations, voir la section Expressions Nameof de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

new, opérateur (Référence C#)

02/11/2020 • 4 minutes to read • Edit Online

L'opérateur `new` crée une nouvelle instance d'un type.

Vous pouvez également utiliser le mot clé `new` comme modificateur de déclaration de membre ou contrainte de type générique.

Appel du constructeur

Pour créer une nouvelle instance d'un type, vous appelez généralement l'un des constructeurs de ce type, à l'aide de l'opérateur `new` :

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Vous pouvez utiliser un initialiseur d'objet ou de collection avec l'opérateur `new` pour instancier et initialiser un objet dans une instruction, comme dans l'exemple suivant :

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

À compter de C# 9,0, les expressions d'appel de constructeur sont de type cible. Autrement dit, si un type cible d'une expression est connu, vous pouvez omettre un nom de type, comme le montre l'exemple suivant :

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

Comme le montre l'exemple précédent, vous utilisez toujours des parenthèses dans une expression de type cible `new` .

Si le type cible d'une `new` expression est inconnu (par exemple, lorsque vous utilisez le `var` mot clé), vous devez spécifier un nom de type.

Création de tableau

Vous utilisez également l'opérateur `new` pour créer une instance de tableau, comme dans l'exemple suivant :

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Utilisez la syntaxe d'initialisation de tableau pour créer une instance de tableau et la remplir avec des éléments dans une instruction. L'exemple suivant montre différentes façons de procéder :

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

Pour plus d'informations sur les tableaux, consultez Tableaux.

Instanciation des types anonymes

Pour créer une instance d'un type anonyme, utilisez l'opérateur `new` et la syntaxe d'initialiseur d'objet :

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destruction des instances de type

Vous n'êtes pas obligé de détruire les instances de type précédemment créées. Les instances de types référence et valeur sont détruites automatiquement. Les instances de types valeur sont détruites dès que le contexte dans lequel elles se trouvent est détruit. Les instances de types référence sont détruites par le garbage collector à une heure non spécifiée après la suppression de la dernière référence.

Pour les instances de type qui contiennent des ressources non managées, par exemple, un descripteur de fichier, il est recommandé d'utiliser le nettoyage déterministe pour s'assurer que les ressources qu'ils contiennent sont libérées dès que possible. Pour plus d'informations, consultez la référence d'API System.IDisposable et l'article using, instruction.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur `new`.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateur new de la spécification du langage C#.

Pour plus d'informations sur une expression de type cible `new` , consultez la Remarque relativeà la proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Initialiseurs d'objets et de collections

sizeof, opérateur (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

L'opérateur `sizeof` retourne le nombre d'octets occupés par une variable d'un type donné. L'argument de l'opérateur `sizeof` doit être le nom d'un type non managé ou d'un paramètre de type qui est contraint comme type non managé.

L'opérateur `sizeof` nécessite un contexte unsafe. Toutefois, les expressions présentées dans le tableau suivant sont évaluées au moment de la compilation par rapport aux valeurs de constante correspondantes et ne nécessitent pas de contexte unsafe :

EXPRESSION	VALEUR CONSTANTE
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

Vous n'avez pas non plus besoin d'utiliser un contexte unsafe quand l'opérande de l'opérateur `sizeof` est le nom d'un type enum.

L'exemple suivant illustre l'utilisation de l'opérateur `sizeof` :

```
using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

L'opérateur `sizeof` retourne un nombre d'octets qui seraient alloués par la Common Language Runtime dans la mémoire managée. Pour les types struct, cette valeur comprend le remplissage, comme le montre l'exemple précédent. Le résultat de l'opérateur `sizeof` peut différer de celui de la méthode Marshal.SizeOf, qui retourne la taille d'un type dans la mémoire *non managée*.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateur sizeof de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs associés au pointeur
- Types de pointeur
- Types liés à la mémoire et à l'étendue
- Génériques en .NET

stackalloc, expression (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Une `stackalloc` expression alloue un bloc de mémoire sur la pile. Un bloc de mémoire alloué dans la pile pendant l'exécution de la méthode est automatiquement supprimé lorsque cette méthode retourne un résultat. Vous ne pouvez pas libérer explicitement la mémoire allouée avec `stackalloc` . Un bloc de mémoire alloué à la pile n'est pas soumis à garbage collection et ne doit pas être épinglé à une `fixed` instruction.

Vous pouvez assigner le résultat d'une `stackalloc` expression à une variable de l'un des types suivants :

- À compter de C# 7,2, System.Span<T> ou System.ReadOnlySpan<T> , comme le montre l'exemple suivant :

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  Vous n'avez pas l'obligation d'utiliser un contexte unsafe lorsque vous affectez un bloc de mémoire alloué dans la pile à une variable Span<T> ou ReadOnlySpan<T>.

  Lorsque vous travaillez avec ces types, vous pouvez utiliser une expression `stackalloc` les expressions conditionnelles ou d'affectation, comme le montre l'exemple suivant :

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
  ```

  À compter de C# 8,0, vous pouvez utiliser une `stackalloc` expression dans d'autres expressions chaque fois qu'une Span<T> ReadOnlySpan<T> variable ou est autorisée, comme le montre l'exemple suivant :

  ```
  Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
  var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6 ,8 });
  Console.WriteLine(ind);  // output: 1
  ```

  > **NOTE**
  >
  > Nous vous recommandons d'utiliser les types Span<T> ou ReadOnlySpan<T> pour travailler avec la mémoire allouée dans la pile quand cela est possible.

- Un type pointeur, comme illustré dans l'exemple suivant :

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

Comme le montre l'exemple précédent, vous devez utiliser un contexte `unsafe` lorsque vous travaillez avec des types pointeurs.

Dans le cas des types pointeur, vous pouvez utiliser une `stackalloc` expression uniquement dans une déclaration de variable locale pour initialiser la variable.

La quantité de mémoire disponible sur la pile est limitée. Si vous allouez une trop grande quantité de mémoire sur la pile, une StackOverflowException est levée. Pour éviter cela, suivez les règles ci-dessous :

- Limitez la quantité de mémoire allouée à l'aide de `stackalloc` :

  ```
  const int MaxStackLimit = 1024;
  Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
  byte[inputLength];
  ```

  Étant donné que la quantité de mémoire disponible sur la pile dépend de l'environnement dans lequel le code est exécuté, soyez prudent lorsque vous définissez la valeur limite réelle.

- Évitez d'utiliser des `stackalloc` boucles internes. Allouez le bloc de mémoire en dehors d'une boucle et réutilisez-le à l'intérieur de la boucle.

Le contenu de la mémoire nouvellement allouée n'est pas défini. Vous devez l'initialiser avant l'utilisation. Par exemple, vous pouvez utiliser la Span<T>.Clear méthode qui affecte à tous les éléments la valeur par défaut de type `T` .

À compter de C# 7,3, vous pouvez utiliser la syntaxe de l'initialiseur de tableau pour définir le contenu de la mémoire nouvellement allouée. L'exemple suivant illustre différentes manières de le faire :

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

Dans Expression `stackalloc T[E]` , `T` doit être un type non managé et `E` doit correspondre à une valeur int non négative.

Sécurité

L'utilisation de `stackalloc` active automatiquement les fonctionnalités de détection des dépassements de mémoire tampon dans le Common Language Runtime (CLR). Si un dépassement de mémoire tampon est détecté, le processus est terminé aussi rapidement que possible pour réduire les risques d'exécution de code malveillant.

spécification du langage C#

Pour plus d'informations, consultez la section allocation de pile de la spécification du langage C# et la fonctionnalité autoriser dans les `stackalloc` contextes imbriqués .

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs associés au pointeur
- Types de pointeur
- Types liés à la mémoire et à l'étendue
- Dos et ne pas de stackalloc

expression Switch (référence C#)

02/11/2020 • 6 minutes to read • Edit Online

Cet article couvre l' `switch` expression, introduite dans C# 8,0. Pour plus d'informations sur l' `switch` instruction, consultez l'article sur l' `switch` instruction dans la section instructions .

Exemple de base

L' `switch` expression fournit des `switch` sémantiques de type like dans un contexte d'expression. Il fournit une syntaxe concise lorsque les bras de commutateur produisent une valeur. L'exemple suivant illustre la structure d'une expression de commutateur. Il convertit les valeurs d'un `enum` qui représente les directions visuelles d'une carte en ligne vers la direction Cardinal correspondante :

```
public static class SwitchExample
{
    public enum Directions
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static void Main()
    {
        var direction = Directions.Right;
        Console.WriteLine($"Map view direction is {direction}");

        var orientation = direction switch
        {
            Directions.Up    => Orientation.North,
            Directions.Right => Orientation.East,
            Directions.Down  => Orientation.South,
            Directions.Left  => Orientation.West,
        };
        Console.WriteLine($"Cardinal orientation is {orientation}");
    }
}
```

L'exemple précédent montre les éléments de base d'une expression de commutateur :

- L' *expression de plage*: l'exemple précédent utilise la variable `direction` en tant qu'expression de plage.
- Les *branches d'expression de commutateur*: chaque ARM d'expression de commutateur contient un *modèle*, un *protecteur de cas*facultatif, le `=>` jeton et une *expression*.

Le résultat de l' *expression de commutateur* est la valeur de l'expression du premier *bras d'expression de commutateur* dont le *modèle* correspond à l' *expression de plage* et dont la protection de la *casse*, si elle est présente, prend la valeur `true` . L' *expression* à droite du `=>` jeton ne peut pas être une instruction d'expression.

Les *branches d'expression de commutateur* sont évaluées dans l'ordre de texte. Le compilateur émet une erreur quand une *expression de commutateur inférieure ARM* ne peut pas être sélectionnée, car une expression de commutateur supérieure *bras* correspond à toutes ses valeurs.

Modèles et protecteurs de cas

De nombreux modèles sont pris en charge dans les branches d'expression de commutateur. L'exemple précédent utilise un *modèle de constante*. Un *modèle de constante* compare l'expression de plage à une valeur. Cette valeur doit être une constante au moment de la compilation. Le *modèle de type* compare l'expression de plage à un type connu. L'exemple suivant récupère le troisième élément d'une séquence. Elle utilise différentes méthodes basées sur le type de la séquence :

```
public static T TypeExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array array => (T)array.GetValue(2),
        IList<T> list      => list[2],
        IEnumerable<T> seq => seq.Skip(2).First(),
    };
```

Les modèles peuvent être récursifs, où un modèle teste un type et si ce type correspond, le modèle correspond à une ou plusieurs valeurs de propriété sur l'expression de plage. Vous pouvez utiliser des modèles récursifs pour étendre l'exemple précédent. Vous ajoutez des branches d'expression de commutateur pour les tableaux qui contiennent moins de 3 éléments. Les modèles récursifs sont illustrés dans l'exemple suivant :

```
public static T RecursiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}       => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array               => (T)array.GetValue(2),
        IList<T> list                    => list[2],
        IEnumerable<T> seq               => seq.Skip(2).First(),
    };
```

Les modèles récursifs peuvent examiner les propriétés de l'expression de plage, mais ne peuvent pas exécuter du code arbitraire. Vous pouvez utiliser une *protection de cas*, spécifiée dans une `when` clause, pour fournir des vérifications similaires pour d'autres types de séquences :

```
public static T CaseGuardExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}                => default(T),
        System.Array { Length : 1} array          => (T)array.GetValue(0),
        System.Array { Length : 2} array          => (T)array.GetValue(1),
        System.Array array                        => (T)array.GetValue(2),
        IEnumerable<T> list when !list.Any()      => default(T),
        IEnumerable<T> list when list.Count() < 3 => list.Last(),
        IList<T> list                             => list[2],
        IEnumerable<T> seq                        => seq.Skip(2).First(),
    };
```

Enfin, vous pouvez ajouter le `_` modèle et le `null` modèle pour intercepter les arguments qui ne sont pas traités par une autre branche d'expression de commutateur. Cela rend l'expression de commutateur *exhaustive*, ce qui signifie que toute valeur possible de l'expression de plage est gérée. L'exemple suivant ajoute ces bras d'expression :

```
public static T ExhaustiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}       => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array               => (T)array.GetValue(2),
        IEnumerable<T> list
            when !list.Any()             => default(T),
        IEnumerable<T> list
            when list.Count() < 3        => list.Last(),
        IList<T> list                    => list[2],
        null                             => throw new ArgumentNullException(nameof(sequence)),
        _                                => sequence.Skip(2).First(),
    };
```

L'exemple précédent ajoute un `null` modèle et remplace le `IEnumerable<T>` modèle de type par un `_` modèle. Le `null` modèle fournit une vérification null en tant qu'expression de commutateur ARM. L'expression pour ce ARM lève une ArgumentNullException . Le `_` modèle correspond à toutes les entrées qui n'ont pas été mises en correspondance avec des bras précédents. Elle doit être postérieure `null` à la vérification ou correspondre aux `null` entrées.

Pour plus d'informations, consultez la proposition de spécifications du langage C# pour les modèles récursifs.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Critères spéciaux

Opérateurs true et false (rférence C#)

02/11/2020 • 4 minutes to read • Edit Online

L' `true` opérateur retourne la bool valeur bool `true` pour indiquer que son opérande a la valeur true. L' `false` opérateur retourne la `bool` valeur `true` pour indiquer que son opérande est définitivement false. Les opérateurs `true` et `false` ne sont pas forcément complémentaires. Autrement dit, les opérateurs `true` et `false` peuvent retourner la valeur `bool``false` pour le même opérande. Si un type définit un des deux opérateurs, il doit aussi définir un autre opérateur.

> **TIP**
> Utilisez le `bool?` type, si vous avez besoin de prendre en charge la logique à trois valeurs (par exemple, lorsque vous utilisez des bases de données qui prennent en charge un type booléen à trois valeurs). C# fournit les opérateurs `&` et `|` qui prennent en charge la logique à trois valeurs avec les opérandes `bool?`. Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Expressions booléennes

Un type avec l'opérateur `true` défini peut être le type de résultat d'une expression conditionnelle de contrôle dans les instructions if, do, while et for et dans l'opérateur conditionnel `?:`. Pour plus d'informations, voir la section Expressions booléennes de la spécification du langage C#.

Opérateurs logiques conditionnels définis par l'utilisateur

Si un type avec les opérateurs `true` et `false` définis surchargel'opérateur OR logique `|` ou l'opérateur AND logique `&` d'une façon, l'opérateur OR logique conditionnel `||` ou l'opérateur AND logique conditionnel `&&`, respectivement, peut être évalué pour les opérandes de ce type. Pour plus d'informations, consultez la section Opérateurs logiques conditionnels définis par l'utilisateur de la spécification du langage C#.

Exemple

L'exemple suivant présente le type qui définit les opérateurs `true` et `false`. Le type surcharge également l'opérateur logique AND `&` de telle sorte que l' `&&` opérateur puisse également être évalué pour les opérandes de ce type.

```
using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

Remarquez le comportement de court-circuit de l'opérateur `&&`. Lorsque la méthode `GetFuelLaunchStatus` retourne `LaunchStatus.Red`, l'opérande de partie droite de l'opérateur `&&` n'est pas évalué. En effet, `LaunchStatus.Red` est false. Le résultat du AND logique ne dépend donc pas de la valeur de l'opérande de partie droite. Voici la sortie de l'exemple :

```
Getting fuel launch status...
Wait!
```

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

with, expression (référence C#)

16/12/2020 • 4 minutes to read • Edit Online

Disponible en C# 9,0 et versions ultérieures, une `with` expression génère une copie de son opérande Record avec les propriétés et les champs spécifiés modifiés :

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }
    }
}
```

Comme le montre l'exemple précédent, vous utilisez la syntaxe de l' initialiseur d'objet pour spécifier les membres à modifier et leurs nouvelles valeurs. Dans une `with` expression, un opérande de gauche doit être d'un type d'enregistrement.

Le résultat d'une `with` expression a le même type au moment de l'exécution que l'opérande de l'expression, comme le montre l'exemple suivant :

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3, Name = A }

    }
}
```

Dans le cas d'un membre de type référence, seule la référence à une instance est copiée lors de la copie d'un enregistrement. La copie et l'enregistrement d'origine ont accès à la même instance de type référence. L'exemple

suivant illustre ce comportement :

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}
```

Tout type d'enregistrement a le *constructeur de copie*. Il s'agit d'un constructeur avec un paramètre unique du type d'enregistrement contenant. Elle copie l'état de son argument dans une nouvelle instance d'enregistrement. Lors de l'évaluation d'une `with` expression, le constructeur de copie est appelé pour instancier une nouvelle instance d'enregistrement basée sur un enregistrement d'origine. Après cela, la nouvelle instance est mise à jour selon les modifications spécifiées. Par défaut, le constructeur de copie est implicite, c'est-à-dire qu'il est généré par le compilateur. Si vous devez personnaliser la sémantique de copie des enregistrements, déclarez explicitement un constructeur de copie avec le comportement souhaité. L'exemple suivant met à jour l'exemple précédent avec un constructeur de copie explicite. Le nouveau comportement de copie consiste à copier des éléments de liste au lieu d'une référence de liste lorsqu'un enregistrement est copié :

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la proposition de fonctionnalité d'enregistrements note:

- `with` formule
- Copier et cloner des membres

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

Surcharge d'opérateur (référence C#)

16/12/2020 • 6 minutes to read • Edit Online

Un type défini par l'utilisateur peut surcharger un opérateur C# prédéfini. Autrement dit, un type peut fournir l'implémentation personnalisée d'une opération dans le cas où un ou les deux opérandes sont de ce type. La section Opérateurs surchargeables indique les opérateurs C# qui peuvent être surchargés.

Utilisez le mot clé `operator` pour déclarer un opérateur. Une déclaration d'opérateur doit respecter les règles suivantes :

- Elle contient un modificateur `public` et un modificateur `static`.
- Un opérateur unaire possède un paramètre d'entrée. Un opérateur binaire a deux paramètres d'entrée. Dans chaque cas, au moins un paramètre doit avoir le type `T` ou `T?`, où `T` est le type qui contient la déclaration d'opérateur.

L'exemple suivant définit une structure simplifiée pour représenter un nombre rationnel. La structure surcharge certains des opérateurs arithmétiques :

```
using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

Vous pouvez étendre l'exemple précédent en définissant une conversion implicite de `int` en `Fraction`. Les opérateurs surchargés devraient alors prendre en charge les arguments de ces deux types. Autrement dit, il deviendrait possible d'ajouter un entier à une fraction et d'obtenir une fraction en conséquence.

Vous utilisez également le mot clé `operator` pour définir une conversion de type personnalisée. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

Opérateurs surchargeables

Le tableau suivant fournit des informations sur la possibilité de surcharge des opérateurs C# :

OPÉRATEURS	POSSIBILITÉ DE SURCHARGE
+x, -x, !x, ~x, ++, --, true, false	Ces opérateurs unaires peuvent être surchargés.
x + y, x-y, x * y, x/y, x% y, x & y, x | y, x ^ y, x << y, x >> y, x = = y, x ! = y, x < y, x > y, x <= y, x > = y	Ces opérateurs binaires peuvent être surchargés. Certains opérateurs doivent être surchargés par paires. Pour plus d'informations, consultez la remarque qui suit ce tableau.
x && y, x || y	Les opérateurs logiques conditionnels ne peuvent pas être surchargés. Toutefois, si un type avec les `true` `false` opérateurs surchargé et surcharge également l' `&` `|` opérateur or d'une certaine façon, l' `&&` `||` opérateur ou, respectivement, peut être évalué pour les opérandes de ce type. Pour plus d'informations, consultez la section Opérateurs logiques conditionnels définis par l'utilisateur de la spécification du langage C#.
une[i], `a?[i]`	L'accès à un élément n'est pas considéré comme un opérateur surchargeable, mais vous pouvez définir un indexeur.
(T) x	L'opérateur de cast ne peut pas être surchargé, mais vous pouvez définir des conversions de type personnalisées qui peuvent être effectuées par une expression de cast. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.
+=, -= , *= , /= , %= , &= , |=, ^= , <<=, >>=	Les opérateurs d'assignation composée ne peuvent pas être surchargés explicitement. Toutefois, quand vous surchargez un opérateur binaire, l'opérateur d'assignation composée correspondant, le cas échéant, est aussi implicitement surchargé. Par exemple, `+=` est évalué à l'aide de l'opérateur `+`, qui peut être surchargé.
^ x, x = y, x. y, `x?.y` , c ? t : f, x ?? y, x ?? = y, x.. y, x->y, => , f (x), As, await, activé, désactivé, par défaut, délégué, est, nameof, New, sizeof, stackalloc, switch, typeof, avec	Ces opérateurs ne peuvent pas être surchargés.

NOTE

Les opérateurs de comparaison doivent être surchargés par paires. Autrement dit, si l'un des opérateurs d'une paire est surchargé, l'autre doit également l'être. Il s'agit de paires telles que les suivantes :

- Opérateurs `==` et `!=`
- Opérateurs `<` et `>`
- Opérateurs `<=` et `>=`

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Surcharge d'opérateur
- Opérateurs

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs de conversion définie par l'utilisateur
- Directives de conception-surcharges d'opérateur
- Directives de conception-opérateurs d'égalité
- Why are overloaded operators always static in C#?

Caractères spéciaux C#

02/11/2020 • 2 minutes to read • Edit Online

Les caractères spéciaux sont des caractères contextuels prédéfinis qui modifient l'élément de programme (chaîne littérale, identificateur ou nom d'attribut) auquel ils sont ajoutés. C# prend en charge les caractères spéciaux suivants :

- @, le caractère d'identificateur textuel.
- $, le caractère de chaîne interpolée.

Voir aussi

- Référence C#
- Guide de programmation C#

$ - interpolation des cordes (référence C)

21/04/2020 • 8 minutes to read • Edit Online

Le caractère spécial `$` identifie un littéral de chaîne comme une *chaîne interpolée*. Une chaîne interpolée est un littéral de chaîne qui peut contenir des *expressions d'interpolation*. Quand une chaîne interpolée est résolue en une chaîne de résultat, les éléments avec des expressions d'interpolation sont remplacés par les représentations sous forme de chaîne des résultats des expressions. Cette fonctionnalité est disponible en commençant par le C 6.

L'interpolation de chaîne offre une syntaxe plus lisible et plus simple pour créer des chaînes mises en forme que la fonctionnalité de mise en forme de chaîne composite. L'exemple suivant utilise les deux fonctionnalités pour produire la même sortie :

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Structure d'une chaîne interpolée

Pour identifier un littéral de chaîne comme chaîne interpolée, préfixez-la du symbole `$`. N'ajoutez pas d'espace entre les signes `$` et `"` au début d'un littéral de chaîne.

La structure d'un élément avec une expression d'interpolation se présente comme suit :

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Les éléments entre crochets sont facultatifs. Le tableau suivant décrit chaque élément :

ÉLÉMENT	DESCRIPTION
`interpolationExpression`	Expression qui produit un résultat à mettre en forme. Représentation des `null` String.Emptycordes de est .
`alignment`	L'expression constante dont la valeur définit le nombre minimum de caractères dans la représentation de chaîne du résultat d'expression. Si le nombre est positif, la représentation sous forme de chaîne est alignée à droite ; s'il est négatif, elle est alignée à gauche. Pour plus d'informations, consultez Composant d'alignement.
`formatString`	Chaîne de format qui est prise en charge par le type de résultat de l'expression. Pour plus d'informations, consultez Composant de chaîne de format.

L'exemple suivant utilise les composants de mise en forme facultatifs décrits ci-dessus :

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left   |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

Caractères spéciaux

Pour ajouter une accolade, « { » ou « } », dans le texte produit par une chaîne interpolée, entrez deux accolades, « {{ » ou « }} ». Pour plus d'informations, consultez Accolades d'échappement.

Comme le caractère deux-points (« : ») a une signification spéciale dans un élément d'expression d'interpolation, pour utiliser un opérateur conditionnel dans une expression d'interpolation, placez cette expression entre parenthèses.

L'exemple suivant montre comment ajouter une accolade dans une chaîne de résultat et comment utiliser un opérateur conditionnel dans une expression d'interpolation :

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

Une chaîne textuelle interpolée `$` commence par `@` le personnage suivi du personnage. Pour plus d'informations sur les chaînes textuelles, consultez les rubriques string et Identificateur textuel.

> **NOTE**
> En commençant par C 8.0, `$` `@` vous pouvez utiliser les `$@"..."` jetons et les jetons dans n'importe quel ordre : les deux et `@$"..."` sont valides interpolés chaînes textuelles. Dans les versions précédentes de C, le `$` jeton doit apparaître devant le `@` jeton.

Conversions implicites et `IFormatProvider` comment spécifier la mise en œuvre

Trois conversions implicites sont possibles à partir d'une chaîne interpolée :

1. Conversion d'une chaîne interpolée en instance de String qui est le résultat de la résolution d'une chaîne interpolée avec des éléments d'expression d'interpolation remplacés par la représentation sous forme de chaîne correctement mise en forme de leurs résultats. Cette conversion CurrentCulture utilise les résultats d'expression de format.

2. Conversion d'une chaîne interpolée en instance FormattableString qui représente une chaîne de format composite avec les résultats d'expression à mettre en forme. Cette conversion vous permet de créer plusieurs chaînes de résultat avec du contenu propre à la culture à partir d'une seule instance de FormattableString. Pour ce faire, appelez l'une des méthodes suivantes :

- Une surcharge ToString() qui produit une chaîne de résultat pour CurrentCulture.
- Une méthode Invariant qui produit une chaîne de résultat pour InvariantCulture.
- Une méthode ToString(IFormatProvider) qui produit une chaîne de résultat pour une culture spécifiée.

Vous pouvez également ToString(IFormatProvider) utiliser la méthode pour fournir IFormatProvider une implémentation définie par l'utilisateur de l'interface qui prend en charge le formatage personnalisé. Pour plus d'informations, voir le formatage personnalisé avec la section ICustomFormatter des types de formatage dans l'article .NET.

3. Conversion d'une chaîne interpolée en instance IFormattable qui vous permet aussi de créer plusieurs chaînes de résultat avec du contenu propre à la culture à partir d'une seule instance de IFormattable.

L'exemple suivant utilise la conversion implicite en FormattableString pour créer des chaînes de résultat propres à la culture :

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Ressources supplémentaires

Si vous ne connaissez pas l'interpolation de chaînes, consultez le tutoriel interactif Interpolation de chaînes en C#. Vous pouvez également vérifier une autre interpolation de chaîne dans le tutoriel Cmd qui démontre comment utiliser des chaînes interpolées pour produire des cordes formatées.

Compilation de chaînes interpolées

Si une chaîne interpolée est de type `string`, elle est généralement transformée en un appel de méthode String.Format. Le compilateur peut remplacer String.Format par String.Concat si le comportement analysé équivaut à une concaténation.

Si une chaîne interpolée est de type IFormattable ou FormattableString, le compilateur génère un appel à la méthode FormattableStringFactory.Create.

spécification du langage C#

Pour plus d'informations, consultez la section Chaînes interpolées de la spécification du langage C#.

Voir aussi

- Référence C#
- Caractères spéciaux C#
- Chaînes
- Chaînes de format numériques standard
- Mise en forme composite
- String.Format

@ (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

Le caractère spécial `@` sert d'identificateur de chaîne textuelle. Il peut être utilisé selon les manières suivantes :

1. Pour activer les mots clés C# à utiliser comme identificateurs. Le caractère `@` est le préfixe d'un élément de code que le compilateur doit interpréter comme un identificateur plutôt qu'un mot clé C#. L'exemple suivant utilise le caractère `@` pour définir un identificateur nommé `for` qu'il utilise dans une boucle `for`.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

2. Pour indiquer qu'un littéral de chaîne doit être interprété textuellement. Le caractère `@` dans cette instance définit un *littéral de chaîne textuelle*. Les séquences d'échappement simples (comme `"\\"` pour une barre oblique inverse), hexadécimales (comme `"\x0041"` pour un A majuscule) et Unicode (comme `"\u0041"` pour un A majuscule), sont interprétées littéralement. Seule une séquence d'échappement de guillemets (`""`) n'est pas interprétée littéralement ; elle produit un guillemet double. En outre, dans le cas d'une chaîne interpolée textuelle, les séquences d'échappement des accolades (`{{` et `}}`) ne sont pas interprétées littéralement ; elles produisent une seule accolade. L'exemple suivant définit deux chemins de fichier identiques, un à l'aide d'un littéral de chaîne standard et l'autre à l'aide d'un littéral de chaîne textuelle. Il s'agit là de l'une des utilisations les plus courantes de littéraux de chaîne textuelle.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//     c:\documents\files\u0066.txt
//     c:\documents\files\u0066.txt
```

   L'exemple suivant illustre l'effet de la définition d'un littéral de chaîne standard et d'un littéral de chaîne textuelle qui contiennent des séquences de caractères identiques.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//     He said, "This is the last chance!"
//     He said, "This is the last \u0063hance\x0021"
```

3. Pour permettre au compilateur de faire la distinction entre les attributs en cas de conflit de noms. Un

attribut est une classe qui dérive de Attribute. Son nom de type comprend généralement le suffixe **Attribute**, bien que le compilateur n'applique pas cette convention. L'attribut peut ensuite être référencé dans le code par son nom de type complet (par exemple, `[InfoAttribute]`) ou son nom abrégé (par exemple, `[Info]`). Toutefois, un conflit survient si deux noms de type d'attribut abrégés sont identiques et que l'un d'eux inclut le suffixe **Attribute**, mais pas l'autre. Par exemple, la compilation du code suivant échoue, car le compilateur ne peut pas déterminer si l'attribut `Info` ou `InfoAttribute` est appliqué à la classe `Example`. Pour plus d'informations, consultez CS1614.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Caractères spéciaux C#

Attributs réservés : attributs de niveau d'assemblage

15/04/2020 • 4 minutes to read • Edit Online

La plupart des attributs sont appliqués à des éléments de langage spécifiques, tels que les classes ou les méthodes. Toutefois, certains attributs sont globaux : ils s'appliquent à la totalité d'un assembly ou d'un module. Par exemple, l'attribut AssemblyVersionAttribute peut être utilisé pour incorporer des informations de version dans un assembly, de la manière suivante :

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Les attributs globaux apparaissent dans le `using` code source après toutes les directives de haut niveau et avant tout type, module ou déclarations d'espace de nom. Les attributs globaux peuvent apparaître dans plusieurs fichiers sources, mais les fichiers doivent être compilés en un seul passage. Visual Studio ajoute des attributs globaux au fichier AssemblyInfo.cs dans les projets cadre .NET. Ces attributs ne sont pas ajoutés aux projets .NET Core.

Les attributs d'assembly sont des valeurs qui fournissent des informations sur un assembly. Ils sont répartis dans les catégories suivantes :

- Attributs d'identité de l'assembly
- Attributs d'informations
- Attributs de manifeste de l'assembly

Attributs d'identité de l'assembly

Trois attributs (avec un nom fort, le cas échéant), déterminent l'identité d'un assembly : nom, version et culture. Ces attributs constituent le nom complet de l'assembly et sont obligatoires quand vous le référencez le code. Vous pouvez définir la version et la culture d'un assembly en utilisant des attributs. Cependant, la valeur du nom est définie par le compilateur, le Visual Studio IDE dans la boîte de dialogue d'information d'assemblage, ou le Lien d'assemblage (Al.exe) lors de la création de l'assemblage. Le nom de l'assemblée est basé sur le manifeste d'assemblage. L'attribut AssemblyFlagsAttribute spécifie si plusieurs copies de l'assembly peuvent coexister.

Le tableau suivant présente les attributs d'identité.

ATTRIBUT	OBJECTIF
AssemblyVersionAttribute	Spécifie la version d'un assembly.
AssemblyCultureAttribute	Spécifie la culture prise en charge par l'assembly.
AssemblyFlagsAttribute	Spécifie si un assembly prend en charge l'exécution côte à côte sur le même ordinateur, dans le même processus ou dans le même domaine d'application.

Attributs d'informations

Vous utilisez des attributs d'information pour fournir des informations supplémentaires sur l'entreprise ou le produit d'un assemblage. Le tableau suivant présente les attributs d'informations définis dans l'espace de noms System.Reflection.

ATTRIBUT	OBJECTIF
AssemblyProductAttribute	Spécifie un nom de produit pour un manifeste d'assemblage.
AssemblyTrademarkAttribute	Spécifie une marque de commerce pour un manifeste d'assemblage.
AssemblyInformationalVersionAttribute	Spécifie une version d'information pour un manifeste d'assemblage.
AssemblyCompanyAttribute	Spécifie un nom d'entreprise pour un manifeste d'assemblage.
AssemblyCopyrightAttribute	Définit un attribut personnalisé qui spécifie un copyright pour un manifeste d'assembly.
AssemblyFileVersionAttribute	Définit un numéro de version spécifique pour la ressource de version de fichier Win32.
CLSCompliantAttribute	Indique si l'assembly est conforme à la spécification CLS (Common Language Specification).

Attributs de manifeste de l'assembly

Vous pouvez utiliser les attributs de manifeste de l'assembly pour fournir des informations dans le manifeste de l'assembly. Les attributs incluent le titre, la description, l'alias par défaut et la configuration. Le tableau suivant présente les attributs de manifeste de l'assembly définis dans l'espace de noms System.Reflection.

ATTRIBUT	OBJECTIF
AssemblyTitleAttribute	Spécifie un titre d'assemblage pour un manifeste d'assemblage.
AssemblyDescriptionAttribute	Spécifie une description d'assemblage pour un manifeste d'assemblage.
AssemblyConfigurationAttribute	Spécifie une configuration d'assemblage (comme la vente au détail ou le débogé) pour un manifeste d'assemblage.
AssemblyDefaultAliasAttribute	Définit un alias par défaut convivial pour un manifeste d'assembly.

Attributs réservés : ConditionalAttribute, ObsoleteAttribute, AttributeUsageAttribute

16/12/2020 • 9 minutes to read • Edit Online

Ces attributs peuvent être appliqués aux éléments de votre code. Ils ajoutent une signification sémantique à ces éléments. Le compilateur utilise ces significations sémantiques pour modifier sa sortie et signaler les erreurs possibles par les développeurs qui utilisent votre code.

Attribut `Conditional`

Avec l'attribut `Conditional`, l'exécution d'une méthode dépend d'un identificateur de prétraitement. L'attribut `Conditional` est un alias pour ConditionalAttribute, et peut être appliqué à une méthode ou une classe d'attributs.

Dans l'exemple suivant, `Conditional` est appliqué à une méthode pour activer ou désactiver l'affichage des informations de diagnostic spécifiques au programme :

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

Si l' `TRACE_ON` identificateur n'est pas défini, la sortie de trace n'est pas affichée. Explorez par vous-même dans la fenêtre interactive.

L' `Conditional` attribut est souvent utilisé avec l' `DEBUG` identificateur pour activer les fonctionnalités de suivi et de journalisation pour les versions Debug, mais pas dans les versions release, comme illustré dans l'exemple suivant :

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

Quand une méthode marquée comme conditionnelle est appelée, la présence ou l'absence du symbole de

prétraitement spécifié détermine si l'appel est inclus ou omis. Si le symbole est défini, l'appel est inclus ; sinon, il est omis. Une méthode conditionnelle doit être une méthode dans une déclaration de classe ou de struct et doit avoir un `void` type de retour. L'utilisation de `Conditional` est plus propre, plus élégante et moins sujette aux erreurs que les méthodes englobantes à l'intérieur des `#if…#endif` blocs.

Si une méthode a plusieurs `Conditional` attributs, un appel à la méthode est inclus si un ou plusieurs symboles conditionnels sont définis (les symboles sont logiquement liés ensemble à l'aide de l'opérateur or). Dans l'exemple suivant, la présence d' `A` un ou de `B` résultats dans un appel de méthode :

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Utiliser `Conditional` avec des classes d'attributs

L'attribut `Conditional` peut également être appliqué à une définition de classe d'attributs. Dans l'exemple suivant, l'attribut personnalisé `Documentation` ajoute uniquement des informations aux métadonnées si `DEBUG` est défini.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Attribut `Obsolete`

L' `Obsolete` attribut marque un élément de code comme n'est plus recommandé pour une utilisation. L'utilisation d'une entité marquée comme obsolète génère un avertissement ou une erreur. `Obsolete` est un attribut à usage unique et peut être appliqué à toute entité qui autorise des attributs. `Obsolete` est un alias pour ObsoleteAttribute.

Dans l'exemple suivant, l' `Obsolete` attribut est appliqué à la classe `A` et à la méthode `B.OldMethod` . Comme le deuxième argument du constructeur d'attribut appliqué à `B.OldMethod` a la valeur `true`, l'utilisation de cette méthode entraîne une erreur du compilateur, alors que l'utilisation de la classe `A` n'entraîne qu'un avertissement. Toutefois, l'appel de `B.NewMethod` ne produit aucun avertissement ni aucune erreur. Par exemple, utilisé avec les définitions précédentes, le code suivant génère deux avertissements et une erreur :

```
using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

La chaîne fournie comme premier argument au constructeur d'attribut sera affichée dans le cadre de l'avertissement ou de l'erreur. Deux avertissements pour la classe `A` sont générés : un pour la déclaration de la référence de classe et un pour le constructeur de classe. L' `Obsolete` attribut peut être utilisé sans arguments, mais il est recommandé d'utiliser à la place une explication.

Attribut `AttributeUsage`

L' `AttributeUsage` attribut détermine la manière dont une classe d'attributs personnalisés peut être utilisée. AttributeUsageAttribute est un attribut que vous appliquez à des définitions d'attribut personnalisé. L'attribut `AttributeUsage` vous permet de contrôler :

- À quels éléments de programme les attributs peuvent être appliqués. Sauf si vous en limitez l'utilisation, un attribut peut être appliqué aux éléments de programme suivants :
  - assembly
  - module
  - field
  - event
  - method
  - param
  - propriété
  - return
  - type
- Si un attribut peut être appliqué plusieurs fois à un même élément de programme.

- Si les attributs sont hérités dans les classes dérivées.

L'exemple suivant montre des paramètres par défaut quand ils sont appliqués explicitement :

```
[AttributeUsage(AttributeTargets.All,
                  AllowMultiple = false,
                  Inherited = true)]
class NewAttribute : Attribute { }
```

Dans cet exemple, la classe `NewAttribute` peut être appliquée à n'importe quel élément de programme pris en charge. Elle ne peut cependant être appliquée qu'une seule fois à chaque entité. L'attribut est hérité par les classes dérivées quand il est appliqué à une classe de base.

Les arguments AllowMultiple et Inherited étant facultatifs, le code suivant a le même effet :

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

Le premier argument AttributeUsageAttribute doit correspondre à un ou plusieurs éléments de l'énumération AttributeTargets. Vous pouvez lier ensemble plusieurs types de cibles avec l'opérateur OR, comme dans l'exemple suivant :

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

À compter de C# 7.3, les attributs peuvent être appliqués à la propriété ou au champ de stockage pour une propriété implémentée automatiquement. L'attribut s'applique à la propriété, sauf si vous spécifiez le spécificateur `field` sur l'attribut. Les deux cas sont illustrés dans l'exemple suivant :

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

Si l'argument AllowMultiple est défini sur `true`, l'attribut résultant peut être appliqué plusieurs fois à une même entité, comme illustré dans l'exemple suivant :

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

Dans ce cas, `MultiUseAttribute` peut être appliqué à plusieurs reprises, car `AllowMultiple` est défini sur `true`. Les deux formats indiqués pour appliquer plusieurs attributs sont valides.

Si Inherited est `false`, l'attribut n'est pas hérité par les classes dérivées d'une classe avec attributs. Par exemple :

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

Dans ce cas, `NonInheritedAttribute` n'est pas appliqué à `DClass` via l'héritage.

Voir aussi

- Attribute
- System.Reflection
- Attributs
- Réflexion

Attributs réservés : Déterminer les informations sur l'appelant

15/04/2020 • 5 minutes to read • Edit Online

En utilisant des attributs d'information, vous obtenez des informations sur l'appelant à une méthode. Vous obtenez le chemin de fichier du code source, le numéro de ligne dans le code source et le nom du membre de l'appelant. Pour obtenir des informations de membre de l'appelant, vous utilisez les attributs qui sont appliqués aux paramètres facultatifs. Chaque paramètre facultatif spécifie une valeur par défaut. Le tableau suivant répertorie les attributs d'informations de l'appelant définis dans l'espace de noms System.Runtime.CompilerServices :

ATTRIBUT	DESCRIPTION	TYPE
CallerFilePathAttribute	Chemin d'accès complet du fichier source qui contient l'appelant. Le chemin complet est le chemin au moment de la compilation.	`String`
CallerLineNumberAttribute	Numéro de ligne dans le fichier source à partir duquel la méthode est appelée.	`Integer`
CallerMemberNameAttribute	Nom de la méthode ou nom de la propriété de l'appelant.	`String`

Ces informations vous aident à écrire des traçages, à débogage et à créer des outils de diagnostic. L'exemple suivant montre comment utiliser les attributs d'info de l'appelant. À chaque appel à la méthode `TraceMessage`, les informations d'appel sont remplacées par des arguments pour les paramètres optionnels.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

Vous spécifiez une valeur par défaut explicite pour chaque paramètre facultatif. Vous ne pouvez pas appliquer des attributs d'information de l'appelant à des paramètres qui ne sont pas spécifiés comme facultatifs. Les attributs d'info de l'appelant ne rendent pas un paramètre facultatif. À la place, ils affectent la valeur par défaut qui est passée si l'argument est oublié. Les valeurs d'info de l'appelant sont émises sous forme d'alphabétisés dans la

langue intermédiaire (IL) au moment de la compilation. Contrairement aux résultats de la propriété StackTrace pour les exceptions, les résultats ne sont pas affectés par l'obfuscation (protection de code). Vous pouvez fournir explicitement les arguments facultatifs pour contrôler ou masquer des informations de l'appelant.

Noms de membres

Vous pouvez utiliser l'attribut `CallerMemberName` pour éviter de spécifier le nom du membre comme argument de `String` à la méthode appelée. Vous évitez ainsi le problème que la **refactorisation de changement de nom** ne modifie pas les valeurs `String`. Cet avantage est particulièrement utile pour les tâches suivantes :

- Utilisation du traçage et des programmes de diagnostic.
- Implémentation de l'interface INotifyPropertyChanged lors de la liaison de données. Cette interface permet à la propriété d'un objet de signaler à un contrôle dépendant que la propriété a été modifiée, afin que ce contrôle puisse afficher les informations mises à jour. Sans attribut `CallerMemberName`, vous devez spécifier le nom de la propriété comme littéral.

Le graphique suivant affiche les noms des membres qui sont retournés lorsque vous utilisez l'attribut `CallerMemberName`.

LES APPELS SE PRODUISENT À L'INTÉRIEUR	RÉSULTAT DE NOM DE MEMBRE
Méthode, propriété ou événement	Le nom de la méthode, la propriété ou l'événement dont l'appel est originaire.
Constructeur	La chaîne « .ctor »
Constructeur statique	La chaîne « .cctor »
Destructeur	La chaîne « finalize »
Opérateurs définis par l'utilisateur ou conversions	Le nom généré pour le membre, par exemple, « op_Addition ».
Constructeur d'attribut	Le nom de la méthode ou de la propriété à laquelle s'applique l'attribut. Si l'attribut est un élément dans un membre (tel qu'un paramètre, une valeur de retour, ou un paramètre de type générique), le résultat est le nom du membre qui est associé à cet élément.
Aucun membre contenant (par exemple, niveau assembly ou attributs qui sont appliqués aux types)	Valeur par défaut du paramètre optionnel.

Voir aussi

- Arguments nommés et facultatifs
- System.Reflection
- Attribute
- Attributs

Les attributs réservés contribuent à l'analyse statique d'État null du compilateur

02/11/2020 • 27 minutes to read • Edit Online

Dans un contexte Nullable, le compilateur effectue une analyse statique du code pour déterminer l'État null de toutes les variables de type référence :

- *not null*: l'analyse statique détermine qu'une valeur non null est assignée à une variable.
- *peut-être null*: l'analyse statique ne peut pas déterminer qu'une valeur non null est assignée à une variable.

Vous pouvez appliquer un certain nombre d'attributs qui fournissent des informations au compilateur sur la sémantique de vos API. Ces informations permettent au compilateur d'effectuer une analyse statique et de déterminer quand une variable n'a pas la valeur null. Cet article fournit une brève description de chacun de ces attributs et explique comment les utiliser. Tous les exemples supposent C# 8,0 ou une version plus récente et le code se trouve dans un contexte Nullable.

Commençons par un exemple familier. Imaginez que votre bibliothèque possède l'API suivante pour récupérer une chaîne de ressource :

```
bool TryGetMessage(string key, out string message)
```

L'exemple précédent suit le `Try*` modèle familier dans .net. Il existe deux arguments de référence pour cette API : `key` et le `message` paramètre. Cette API comporte les règles suivantes relatives à la valeur null de ces arguments :

- Les appelants ne doivent pas passer `null` comme argument pour `key` .
- Les appelants peuvent passer une variable dont la valeur est `null` l'argument pour `message` .
- Si la `TryGetMessage` méthode retourne `true` , la valeur de `message` n'est pas null. Si la valeur de retour est `false,` la valeur de `message` (et son état null) est null.

La règle pour `key` peut être exprimée par le type de variable : `key` doit être un type référence qui n'accepte pas les valeurs NULL. Le `message` paramètre est plus complexe. Elle autorise `null` comme argument, mais garantit que, en cas de réussite, cet `out` argument n'est pas null. Pour ces scénarios, vous avez besoin d'un vocabulaire plus riche pour décrire les attentes.

Plusieurs attributs ont été ajoutés pour exprimer des informations supplémentaires sur l'État null des variables. Tout le code que vous avez écrit avant C# 8 a introduit des types de référence Nullable était *null oublie*. Cela signifie que toute variable de type référence peut avoir la valeur null, mais que les vérifications de valeur NULL ne sont pas requises. Une fois que votre code prend en *charge les valeurs NULL*, ces règles changent. Les types référence ne doivent jamais être la `null` valeur, et les types référence Nullable doivent être vérifiés `null` avant d'être déréférencés.

Les règles de vos API sont probablement plus compliquées, comme vous l'avez vu dans le `TryGetValue` scénario d'API. La plupart de vos API ont des règles plus complexes pour le moment où les variables peuvent ou ne peuvent pas être `null` . Dans ce cas, vous allez utiliser l'un des attributs suivants pour exprimer ces règles :

- AllowNull a: un argument d'entrée qui n'accepte pas les valeurs NULL peut être null.
- DisallowNull: un argument d'entrée Nullable ne doit jamais avoir la valeur null.
- MaybeNull: une valeur de retour qui n'accepte pas les valeurs NULL peut être null.

- NOTNULL: une valeur de retour Nullable ne sera jamais null.
- MaybeNullWhen: un argument d'entrée qui n'accepte pas les valeurs NULL peut être NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullWhen: un argument d'entrée Nullable ne sera pas NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullIfNotNull: une valeur de retour n'est pas null si l'argument pour le paramètre spécifié n'est pas null.
- DoesNotReturn: une méthode ne retourne jamais. En d'autres termes, elle lève toujours une exception.
- DoesNotReturnIf: cette méthode ne retourne jamais si le `bool` paramètre associé a la valeur spécifiée.

Les descriptions précédentes sont une référence rapide à ce que fait chaque attribut. Chaque section suivante décrit le comportement et la signification de manière plus approfondie.

L'ajout de ces attributs donne au compilateur plus d'informations sur les règles de votre API. Quand le code appelant est compilé dans un contexte compatible avec la valeur null, le compilateur avertit les appelants lorsqu'ils enfreignent ces règles. Ces attributs n'autorisent pas de vérifications supplémentaires sur votre implémentation.

Spécifier les conditions préalables : `AllowNull` et `DisallowNull`

Prenons l'exemple d'une propriété en lecture/écriture qui n'est jamais retournée `null` , car elle a une valeur par défaut raisonnable. Les appelants passent `null` à l'accesseur Set lors de la définition de cette valeur par défaut. Par exemple, imaginez un système de messagerie qui demande un nom d'écran dans une salle de conversation. Si aucune valeur n'est fournie, le système génère un nom aléatoire :

```
public string ScreenName
{
   get => _screenName;
   set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

Quand vous compilez le code précédent dans un contexte oublie Nullable, tout est parfait. Une fois que vous activez les types référence Nullable, la `ScreenName` propriété devient une référence qui n'accepte pas les valeurs NULL. C'est correct pour l' `get` accesseur : il ne retourne jamais `null` . Les appelants n'ont pas besoin de vérifier la propriété retournée pour `null` . Mais maintenant, le paramétrage de la propriété pour `null` générer un avertissement. Pour continuer à prendre en charge ce type de code, vous ajoutez l' System.Diagnostics.CodeAnalysis.AllowNullAttribute attribut à la propriété, comme illustré dans le code suivant :

```
[AllowNull]
public string ScreenName
{
   get => _screenName;
   set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

Vous devrez peut-être ajouter une `using` directive pour System.Diagnostics.CodeAnalysis utiliser ce et d'autres attributs abordés dans cet article. L'attribut est appliqué à la propriété, et non à l' `set` accesseur. L' `AllowNull` attribut spécifie des *conditions préalables*et s'applique uniquement aux entrées. L' `get` accesseur a une valeur de retour, mais aucun argument d'entrée. Par conséquent, l' `AllowNull` attribut s'applique uniquement à l' `set` accesseur.

L'exemple précédent montre ce qu'il faut rechercher lors de l'ajout `AllowNull` de l'attribut à un argument :

1. Le contrat général pour cette variable est qu'elle ne doit pas l'être `null` , donc vous souhaitez un type de référence non Nullable.
2. Il existe des scénarios pour la variable d'entrée `null` , bien qu'il ne s'agisse pas de l'utilisation la plus courante.

La plupart du temps, vous aurez besoin de cet attribut pour les propriétés, ou `in` , `out` et les `ref` arguments. L' `AllowNull` attribut est le meilleur choix lorsqu'une variable est généralement non null, mais que vous devez autoriser `null` comme condition préalable.

À l'inverse, avec les scénarios d'utilisation de `DisallowNull` :, vous utilisez cet attribut pour spécifier qu'une variable d'entrée d'un type de référence Nullable ne doit pas être `null` . Considérez une propriété où `null` est la valeur par défaut, mais les clients peuvent uniquement la définir sur une valeur non null. Considérez le code suivant :

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

Le code précédent est la meilleure façon d'exprimer votre conception de `ReviewComment` `null` , mais ne peut pas avoir la valeur `null` . Une fois que ce code prend en charge les valeurs NULL, vous pouvez exprimer ce concept plus clairement aux appelants à l'aide du System.Diagnostics.CodeAnalysis.DisallowNullAttribute :

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

Dans un contexte Nullable, l' `ReviewComment` `get` accesseur peut retourner la valeur par défaut de `null` . Le compilateur vous avertit qu'il doit être vérifié avant l'accès. En outre, il avertit les appelants que, même s'ils peuvent l'être `null` , les appelants ne doivent pas lui affecter explicitement la valeur `null` . L' `DisallowNull` attribut spécifie également une *condition préalable*, il n'affecte pas l' `get` accesseur. Vous utilisez l' `DisallowNull` attribut lorsque vous observez les caractéristiques suivantes :

1. La variable peut être `null` dans des scénarios de base, souvent lors de la première instanciation.
2. La variable ne doit pas avoir explicitement la valeur `null` .

Ces situations sont courantes dans le code qui a été initialement *null oublie*. Les propriétés de l'objet sont peut-être définies dans deux opérations d'initialisation distinctes. Il se peut que certaines propriétés soient définies uniquement après la fin d'un travail asynchrone.

Les `AllowNull` `DisallowNull` attributs et vous permettent de spécifier que les conditions préalables sur les variables peuvent ne pas correspondre aux annotations Nullable sur ces variables. Celles-ci fournissent des informations plus détaillées sur les caractéristiques de votre API. Ces informations supplémentaires permettent aux appelants d'utiliser correctement votre API. N'oubliez pas de spécifier des conditions préalables à l'aide des attributs suivants :

- AllowNull a: un argument d'entrée qui n'accepte pas les valeurs NULL peut être null.
- DisallowNull: un argument d'entrée Nullable ne doit jamais avoir la valeur null.

Spécifier les conditions postérieures : `MaybeNull` et `NotNull`

Supposons que vous avez une méthode avec la signature suivante :

```
public Customer FindCustomer(string lastName, string firstName)
```

Vous avez probablement écrit une méthode comme celle-ci pour retourner `null` lorsque le nom recherché n'a pas été trouvé. `null` Indique clairement que l'enregistrement n'a pas été trouvé. Dans cet exemple, vous allez probablement modifier le type de retour de `Customer` en `Customer?` . La déclaration de la valeur de retour en tant que type référence Nullable spécifie clairement l'objectif de cette API.

Pour les raisons couvertes par les définitions génériques et la possibilité de valeur null , cette technique ne fonctionne pas avec les méthodes génériques. Vous pouvez avoir une méthode générique qui suit un modèle similaire :

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

Vous ne pouvez pas spécifier que la valeur de retour est `T?` . La méthode retourne `null` lorsque l'élément recherché est introuvable. Étant donné que vous ne pouvez pas déclarer un `T?` type de retour, vous ajoutez l' `MaybeNull` annotation à la méthode Return :

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

Le code précédent informe les appelants que le contrat implique un type non Nullable, mais la valeur de retour *peut* en fait être null. Utilisez l' `MaybeNull` attribut lorsque votre API doit être un type non Nullable, en général un paramètre de type générique, mais il peut y avoir des instances où `null` serait retourné.

Vous pouvez également spécifier qu'une valeur de retour ou `out` un `ref` argument ou n'est pas null bien que le type soit un type référence Nullable. Prenons l'exemple d'une méthode qui garantit qu'un tableau est suffisamment grand pour contenir un certain nombre d'éléments. Si l'argument d'entrée n'a pas de capacité, la routine alloue un nouveau tableau et copie tous les éléments existants dans celui-ci. Si l'argument d'entrée est `null` , la routine allouera un nouveau stockage. Si la capacité est suffisante, la routine ne fait rien :

```
public void EnsureCapacity<T>(ref T[] storage, int size)
```

Vous pouvez appeler cette routine comme suit :

```
// messages has the default value (null) when EnsureCapacity is called:
EnsureCapacity<string>(ref messages, 10);
// messages is not null.
EnsureCapacity<string>(messages, 50);
```

Après avoir activé les types référence null, vous souhaitez vous assurer que le code précédent se compile sans avertissements. Lorsque la méthode retourne, l' `storage` argument est garanti comme non null. Toutefois, il est acceptable d'appeler `EnsureCapacity` avec une référence null. Vous pouvez créer `storage` un type référence Nullable et ajouter la `NotNull` condition postérieure à la déclaration du paramètre :

```
public void EnsureCapacity<T>([NotNull] ref T[]? storage, int size)
```

Le code précédent exprime clairement le contrat existant : les appelants peuvent passer une variable avec la

`null` valeur, mais la valeur de retour est garantie pour ne jamais avoir la valeur null. L' `NotNull` attribut est particulièrement utile pour `ref` les `out` arguments et `null` , où peut être passé comme argument, mais que cet argument est garanti comme non NULL lorsque la méthode est retournée.

Vous spécifiez des post-conditions inconditionnelles à l'aide des attributs suivants :

- MaybeNull: une valeur de retour qui n'accepte pas les valeurs NULL peut être null.
- NOTNULL: une valeur de retour Nullable ne sera jamais null.

Spécifier les conditions de publication conditionnelles : `NotNullWhen` , `MaybeNullWhen` et `NotNullIfNotNull`

Vous êtes probablement familiarisé avec la `string` méthode String.IsNullOrEmpty(String) . Cette méthode retourne `true` lorsque l'argument a la valeur null ou est une chaîne vide. Il s'agit d'une forme de contrôle de valeur NULL : les appelants n'ont pas besoin de vérifier la valeur null si la méthode retourne `false` . Pour rendre une méthode comme cette prise en charge de Nullable, vous devez définir l'argument sur un type de référence Nullable et ajouter l' `NotNullWhen` attribut :

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value);
```

Cela indique au compilateur que tout code dont la valeur de retour est n'a `false` pas besoin d'être vérifié par null. L'ajout de l'attribut informe l'analyse statique du compilateur qui `IsNullOrEmpty` effectue la vérification de la valeur null nécessaire : lorsqu'il retourne `false` , l'argument d'entrée n'est pas `null` .

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
   int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

La String.IsNullOrEmpty(String) méthode sera annotée comme indiqué ci-dessus pour .net Core 3,0. Vous pouvez avoir des méthodes similaires dans votre code base, qui vérifient l'état des objets pour les valeurs NULL. Le compilateur ne reconnaîtra pas les méthodes de vérification null personnalisées, et vous devrez ajouter vous-même les annotations. Lorsque vous ajoutez l'attribut, l'analyse statique du compilateur sait quand la variable testée a été vérifiée avec la valeur null.

Une autre utilisation de ces attributs est le `Try*` modèle. Les post-conditions pour les `ref` `out` variables et sont communiquées par le biais de la valeur de retour. Prenons l'exemple de cette méthode ci-dessus :

```
bool TryGetMessage(string key, out string message)
```

La méthode précédente suit un idiome .NET typique : la valeur de retour indique si `message` a été défini sur la valeur trouvée ou, si aucun message n'est trouvé, à la valeur par défaut. Si la méthode retourne `true` , la valeur de `message` n'est pas NULL ; sinon, la méthode affecte la valeur `message` null.

Vous pouvez communiquer cet idiome à l'aide de l' `NotNullWhen` attribut. Quand vous mettez à jour la signature pour les types de référence Nullable, créez `message` un `string?` et ajoutez un attribut :

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
```

Dans l'exemple précédent, la valeur de `message` est dite NOT NULL lorsque `TryGetMessage` retourne `true` .

Vous devez annoter des méthodes similaires dans votre code base de la même façon : les arguments peuvent être `null` , et sont connus pour ne pas avoir la valeur NULL lorsque la méthode est retournée `true` .

Vous pouvez également avoir un attribut final. Parfois, l'État null d'une valeur de retour dépend de l'État null d'un ou plusieurs arguments d'entrée. Ces méthodes retournent une valeur non null chaque fois que certains arguments d'entrée ne sont pas `null` . Pour annoter correctement ces méthodes, vous utilisez l' `NotNullIfNotNull` attribut. Considérez la méthode suivante :

```
string GetTopLevelDomainFromFullUrl(string url);
```

Si l' `url` argument n'est pas null, la sortie n'est pas `null` . Une fois que les références Nullable sont activées, cette signature fonctionne correctement, à condition que votre API n'accepte jamais une entrée null. Toutefois, si l'entrée peut être null, la valeur de retour peut également être null. Par conséquent, vous pouvez remplacer la signature par le code suivant :

```
string? GetTopLevelDomainFromFullUrl(string? url);
```

Cela fonctionne également, mais obligent souvent les appelants à implémenter des `null` vérifications supplémentaires. Le contrat est que la valeur de retour serait `null` uniquement lorsque l'argument d'entrée `url` est `null` . Pour exprimer ce contrat, vous devez annoter cette méthode comme indiqué dans le code suivant :

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url);
```

La valeur de retour et l'argument ont tous deux été annotés avec le `?` qui indique que peut être `null` . L'attribut clarifie davantage que la valeur de retour n'est pas NULL lorsque l' `url` argument n'est pas `null` .

Vous spécifiez des post-conditions conditionnelles à l'aide des attributs suivants :

- MaybeNullWhen: un argument d'entrée qui n'accepte pas les valeurs NULL peut être NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullWhen: un argument d'entrée Nullable ne sera pas NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullIfNotNull: une valeur de retour n'est pas null si l'argument d'entrée pour le paramètre spécifié n'est pas null.

Vérifier le code inaccessible

Certaines méthodes, généralement des applications d'assistance d'exception ou d'autres méthodes utilitaires, se ferment toujours en levant une exception. Ou bien, une application d'assistance peut lever une exception en fonction de la valeur d'un argument booléen.

Dans le premier cas, vous pouvez ajouter l' `DoesNotReturn` attribut à la déclaration de méthode. Le compilateur vous aide de trois façons. Tout d'abord, le compilateur émet un avertissement s'il existe un chemin d'accès où la méthode peut se fermer sans lever d'exception. Deuxièmement, le compilateur marque tout code après un appel à cette méthode comme étant *inaccessible*, jusqu'à ce qu'une `catch` clause appropriée soit rencontrée. Troisièmement, le code inaccessible n'affecte pas les États null. Prenons la méthode suivante :

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (!isInitialized)
    {
        FailFast();
    }

    // unreachable code:
    _field = containedField;
}
```

Dans le deuxième cas, vous ajoutez l' `DoesNotReturnIf` attribut à un paramètre booléen de la méthode. Vous pouvez modifier l'exemple précédent comme suit :

```
private void FailFast([DoesNotReturnIf(false)] bool isValid)
{
    if (!isValid)
    {
        throw new InvalidOperationException();
    }
}

public void SetState(object containedField)
{
    FailFast(isInitialized);

    // unreachable code when "isInitialized" is false:
    _field = containedField;
}
```

Résumé

> **IMPORTANT**
>
> La documentation officielle suit la dernière version de C#. Nous écrivons actuellement pour C# 9,0. Selon la version de C# que vous utilisez, il se peut que diverses fonctionnalités ne soient pas disponibles. La version C# *par défaut* de votre projet est basée sur la version cible de .NET Framework. Pour plus d'informations, consultez paramètres par défautde contrôle de version du langage C#.

L'ajout de types de référence Nullable fournit un vocabulaire initial pour décrire les attentes des API pour les variables qui pourraient être `null` . Les attributs supplémentaires fournissent un vocabulaire plus riche pour décrire l'État null des variables en tant que conditions préalables et post-conditions. Ces attributs décrivent plus clairement vos attentes et offrent une meilleure expérience aux développeurs qui utilisent vos API.

Lorsque vous mettez à jour des bibliothèques pour un contexte Nullable, ajoutez ces attributs pour guider les utilisateurs de vos API vers l'utilisation correcte. Ces attributs vous aident à décrire complètement l'État null des arguments d'entrée et les valeurs de retour :

- AllowNull a: un argument d'entrée qui n'accepte pas les valeurs NULL peut être null.
- DisallowNull: un argument d'entrée Nullable ne doit jamais avoir la valeur null.
- MaybeNull: une valeur de retour qui n'accepte pas les valeurs NULL peut être null.

- NOTNULL: une valeur de retour Nullable ne sera jamais null.
- MaybeNullWhen: un argument d'entrée qui n'accepte pas les valeurs NULL peut être NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullWhen: un argument d'entrée Nullable ne sera pas NULL lorsque la méthode retourne la `bool` valeur spécifiée.
- NotNullIfNotNull: une valeur de retour n'est pas null si l'argument d'entrée pour le paramètre spécifié n'est pas null.
- DoesNotReturn: une méthode ne retourne jamais. En d'autres termes, elle lève toujours une exception.
- DoesNotReturnIf: cette méthode ne retourne jamais si le `bool` paramètre associé a la valeur spécifiée.

Directives de préprocesseur C#

16/12/2020 • 2 minutes to read • Edit Online

Cette section contient des informations sur les directives de préprocesseur C# :

- #if
- #else
- #elif
- #endif
- #define
- #undef
- #warning
- #error
- #line
- #nullable
- #region
- #endregion
- #pragma
- AVERTISSEMENT #pragma
- somme de contrôle #pragma

Pour plus d'informations et des exemples, consultez les rubriques spécifiques.

Bien que le compilateur n'ait pas de préprocesseur distinct, les directives décrites dans cette section sont traitées comme si c'était le cas. Elles sont utilisées pour faciliter la compilation conditionnelle. Contrairement aux directives C et C++, vous ne pouvez pas les utiliser pour créer des macros.

Une directive de préprocesseur doit être la seule instruction sur une ligne.

Voir aussi

- Référence C#
- Guide de programmation C#

#if (référence C#)

02/11/2020 • 5 minutes to read • Edit Online

Quand le compilateur C# rencontre une directive `#if`, suivie éventuellement d'une directive #endif, il compile le code entre les directives uniquement si le symbole spécifié est défini. Contrairement à C et C++, vous ne pouvez pas attribuer de valeur numérique à un symbole. L' `#if` instruction en C# est booléenne et teste uniquement si le symbole a été défini ou non. Par exemple :

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Vous pouvez utiliser les opérateurs == (égalité) et ! = (inégalité) uniquement pour tester les valeurs bool `true` ou `false` . `true` signifie que le symbole est défini. L'instruction `#if DEBUG` a la même signification que `#if (DEBUG == true)` . Vous pouvez utiliser les && (and), || (ou)et ! (not) opérateurs permettant d'évaluer si plusieurs symboles ont été définis. Vous pouvez également regrouper des symboles et des opérateurs à l'aide de parenthèses.

Notes

`#if`, ainsi que les directives #else, #elif, #endif, #defineet #undef , vous permettent d'inclure ou d'exclure du code en fonction de l'existence d'un ou plusieurs symboles. Cela peut être utile lors de la compilation du code pour une version Debug ou lors de la compilation d'une configuration spécifique.

Une directive conditionnelle commençant par une directive `#if` doit se terminer explicitement par une directive `#endif`.

`#define` vous permet de définir un symbole. En utilisant ensuite ce dernier comme expression passée à la directive `#if`, l'expression correspond à `true`.

Vous pouvez également définir un symbole avec l'option de compilateur-define . Vous pouvez annuler la définition d'un symbole avec #undef.

Un symbole que vous définissez avec `-define` ou `#define` n'est pas en conflit avec une variable du même nom. Autrement dit, un nom de variable ne doit pas être passé à une directive de préprocesseur, et un symbole ne peut être évalué que par une directive de préprocesseur.

La portée d'un symbole créé avec `#define` est le fichier dans lequel il a été défini.

Le système de génération est également conscient des symboles de préprocesseur prédéfinis qui représentent différents frameworks cibles dans les projets de type SDK. Elles sont utiles lors de la création d'applications pouvant cibler plusieurs versions de .NET.

VERSIONS CIBLES DE .NET FRAMEWORK	SYMBOLES
.NET Framework	`NETFRAMEWORK` , `NET48` , `NET472` , `NET471` , `NET47` , `NET462` , `NET461` , `NET46` , `NET452` , `NET451` , `NET45` , `NET40` , `NET35` , `NET20`

VERSIONS CIBLES DE .NET FRAMEWORK	SYMBOLES
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`, `NETSTANDARD1_1`, `NETSTANDARD1_0`
.NET 5 (et .NET Core)	`NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`

NOTE

Pour les projets traditionnels de type non-SDK, vous devez configurer manuellement les symboles de compilation conditionnelle pour les différents frameworks cibles dans Visual Studio via les pages de propriétés du projet.

Parmi les autres symboles prédéfinis se trouvent les constantes DEBUG et TRACE. Vous pouvez remplacer les valeurs définies pour le projet à l'aide de `#define`. Le symbole DEBUG, par exemple, est automatiquement défini en fonction de vos propriétés de configuration de build (mode « Debug » ou « Release »).

Exemples

L'exemple suivant montre comment définir un symbole MYTEST sur un fichier, puis tester les valeurs des symboles MYTEST et DEBUG. La sortie de cet exemple dépend du mode de configuration que vous avez utilisé pour créer le projet (Debug ou Release).

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

L'exemple suivant montre comment tester différents frameworks cibles pour pouvoir utiliser les nouvelles API dans la mesure du possible :

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#
- Procédure : Effectuer une compilation conditionnelle avec Trace et Debug

#else (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#else` vous permet de créer une directive conditionnelle composée de sorte que si aucune des expressions contenues dans les précédentes directives #if ou #elif (facultatif) n'a la valeur `true`, le compilateur évalue l'ensemble du code entre `#else` et la directive `#endif` suivante.

Notes

#endif doit être la directive de préprocesseur suivante après `#else`. Consultez #if pour obtenir un exemple d'utilisation de `#else`.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#elif (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#elif` vous permet de créer une directive conditionnelle composée. L'expression `#elif` est évaluée si ni l'expression #if précédente ni une expression de directive `#elif` facultative précédente n'est évaluée à `true`. Si une expression `#elif` est évaluée à `true`, le compilateur évalue l'ensemble du code situé entre `#elif` et la directive conditionnelle suivante. Par exemple :

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

Vous pouvez utiliser les opérateurs `==` (égalité), `!=` (inégalité) `&&` (et) et `||` (ou) pour évaluer plusieurs symboles. Vous pouvez également regrouper des symboles et des opérateurs à l'aide de parenthèses.

Notes

`#elif` revient à utiliser :

```
#else
#if
```

`#elif` est plus simple à utiliser, car chaque expression `#if` a besoin d'une expression #endif, alors qu'une expression `#elif` peut être utilisée sans expression `#endif` correspondante.

Consultez #if pour obtenir un exemple d'utilisation de `#elif`.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#endif (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#endif` spécifie la fin d'une directive conditionnelle, qui a commencé avec la directive #if. Par exemple,

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Notes

Une directive conditionnelle commençant par une directive `#if` doit se terminer explicitement par une directive `#endif`. Consultez #if pour obtenir un exemple d'utilisation de `#endif`.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#define (référence C#)

02/11/2020 • 3 minutes to read • Edit Online

Vous utilisez `#define` pour définir un symbole. Quand vous utilisez le symbole sous forme d'expression passée à la directive #if, l'expression donne la valeur `true`, comme illustré dans l'exemple suivant :

```
#define DEBUG
```

Notes

> **NOTE**
> La directive `#define` ne peut pas être utilisée pour déclarer des valeurs constantes comme c'est le cas en général en C et C++. Les constantes en C# sont définies plutôt comme des membres statiques d'une classe ou d'un struct. Si vous avez plusieurs constantes de ce type, créez une classe « Constantes » distincte pour les stocker.

Les symboles peuvent être utilisés pour spécifier des conditions de compilation. Vous pouvez tester le symbole avec #if ou #elif. Vous pouvez également utiliser ConditionalAttribute pour effectuer une compilation conditionnelle.

Vous pouvez définir un symbole, mais vous ne pouvez pas attribuer de valeur à un symbole. La directive `#define` doit apparaître dans le fichier avant l'utilisation d'instructions qui ne sont pas également des directives de préprocesseur.

Vous pouvez également définir un symbole avec l'option de compilateur-define . Vous pouvez annuler la définition d'un symbole avec #undef.

Un symbole que vous définissez avec `-define` ou `#define` n'est pas en conflit avec une variable du même nom. Autrement dit, un nom de variable ne doit pas être passé à une directive de préprocesseur et un symbole ne peut être évalué que par une directive de préprocesseur.

La portée d'un symbole qui a été créé à l'aide de `#define` est le fichier dans lequel le symbole a été défini.

Comme le montre l'exemple suivant, vous devez placer les directives `#define` en haut du fichier.

```
#define DEBUG
//#define TRACE
#undef TRACE

using System;

public class TestDefine
{
    static void Main()
    {
#if (DEBUG)
        Console.WriteLine("Debugging is enabled.");
#endif

#if (TRACE)
     Console.WriteLine("Tracing is enabled.");
#endif
    }
}
// Output:
// Debugging is enabled.
```

Pour obtenir un exemple montrant comment annuler la définition d'un symbole, consultez #undef.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#
- const
- Procédure : Effectuer une compilation conditionnelle avec Trace et Debug
- #undef
- #if

#undef (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

`#undef` vous permet d'annuler la définition d'un symbole de telle sorte qu'en utilisant le symbole comme expression dans une directive #if, l'expression correspond à `false`.

Un symbole peut être défini à l'aide de la directive #define ou de l'option de compilateur-define . La directive `#undef` doit figurer dans le fichier préalablement à l'utilisation d'instructions qui ne sont pas aussi des directives.

Exemple

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
#if DEBUG
        Console.WriteLine("DEBUG is defined");
#else
        Console.WriteLine("DEBUG is not defined");
#endif
    }
}
```

DEBUG n'est pas défini

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#warning (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#warning` vous permet de générer un avertissement du compilateur de premier niveau CS1030 à partir d'un emplacement spécifique dans votre code. Par exemple :

```
#warning Deprecated code in this method.
```

Notes

`#warning` est souvent utilisé dans une directive conditionnelle. Il est aussi possible de générer une erreur définie par l'utilisateur avec #error.

Exemple

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#warning DEBUG is defined
#endif
    }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#error (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#error` vous permet de générer une erreur définie par l'utilisateur CS1029 à partir d'un emplacement spécifique dans votre code. Par exemple :

```
#error Deprecated code in this method.
```

> **NOTE**
> Le compilateur traite `#error version` de façon spéciale et signale une erreur du compilateur, CS8304, avec un message contenant le compilateur utilisé et les versions de langage.

Remarques

`#error` est souvent utilisé dans une directive conditionnelle.

Il est possible aussi de générer un avertissement défini par l'utilisateur avec #warning.

Exemple

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#error DEBUG is defined
#endif
    }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#line (référence C#)

02/11/2020 • 4 minutes to read • Edit Online

`#line` vous permet de changer la numérotation de lignes du compilateur et (éventuellement) le nom de fichier pour les erreurs et les avertissements.

L'exemple suivant montre comment signaler deux avertissements associés à des numéros de ligne. La directive `#line 200` assigne de force le numéro 200 à la ligne suivante (bien que le numéro par défaut soit 6) ; jusqu'à la prochaine directive `#line`, le nom de fichier indiqué est « Special ». La directive `#line default` rétablit la numérotation de lignes par défaut, qui compte les lignes renumérotées par la directive précédente.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

La compilation produit la sortie suivante :

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

Remarques

La directive `#line` peut être utilisée dans une étape intermédiaire automatisée du processus de génération. Par exemple, si des lignes ont été supprimées du fichier de code source d'origine, mais que vous voulez que le compilateur continue de générer une sortie sur la base de la numérotation de lignes d'origine du fichier, vous pouvez supprimer les lignes et simuler ensuite la numérotation de lignes d'origine avec `#line`.

La directive `#line hidden` masque les lignes successives du débogueur, de sorte que quand le développeur parcourt le code, les lignes situées entre une directive `#line hidden` et la directive `#line` suivante (en supposant qu'il ne s'agit pas d'une autre directive `#line hidden`) sont ignorées. Cette option peut aussi être utilisée pour permettre à ASP.NET de différencier le code défini par l'utilisateur du code généré par l'ordinateur. Même si ASP.NET est le principal utilisateur de cette fonctionnalité, il est probable que d'autres générateurs de code source pourront l'utiliser.

Une directive `#line hidden` n'affecte ni les noms de fichiers ni les numéros de lignes dans les rapports d'erreurs. Autrement dit, si une erreur se produit dans un bloc masqué, le compilateur indique le nom du fichier et le numéro de ligne actuels de l'erreur.

La directive `#line filename` spécifie le nom de fichier que vous souhaitez voir apparaître dans la sortie du compilateur. Par défaut, le nom réel du fichier de code source est utilisé. Le nom de fichier doit être entre guillemets doubles (" ") et doit être précédé d'un numéro de ligne.

Un fichier de code source peut contenir un nombre illimité de directives `#line`.

Exemple 1

L'exemple suivant montre comment le débogueur ignore les lignes masquées dans le code. Quand vous exécutez l'exemple, celui-ci affiche trois lignes de texte. Or, quand vous définissez un point d'arrêt, comme dans l'exemple, et que vous appuyez sur F10 pour parcourir le code, vous pouvez remarquer que le débogueur ignore la ligne masquée. Vous remarquerez aussi que même si vous définissez un point d'arrêt au niveau de la ligne masquée, le débogueur continuera de l'ignorer.

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#nullable (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

La `#nullable` directive de préprocesseur définit le contexte d' *annotation Nullable* et le *contexte d'avertissement Nullable*. Cette directive contrôle si les annotations Nullable ont un effet et si des avertissements de possibilité de valeur NULL sont fournis. Chaque contexte est *désactivé* ou *activé*.

Les deux contextes peuvent être spécifiés au niveau du projet (en dehors du code source C#). La `#nullable` directive contrôle les contextes d'annotation et d'avertissement et est prioritaire sur les paramètres au niveau du projet. Une directive définit le ou les contexte (s) qu'elle contrôle jusqu'à ce qu'une autre directive la remplace, ou jusqu'à la fin du fichier source.

L'effet des directives est le suivant :

- `#nullable disable` : Définit l'annotation Nullable et les contextes d'avertissement sur *désactivé*.
- `#nullable enable` : Définit l'annotation Nullable et les contextes d'avertissement sur *activé*.
- `#nullable restore` : Restaure l'annotation Nullable et les contextes d'avertissement dans les paramètres du projet.
- `#nullable disable annotations` : Définit le contexte d'annotation Nullable sur *désactivé*.
- `#nullable enable annotations` : Définit le contexte d'annotation Nullable sur *activé*.
- `#nullable restore annotations` : Restaure le contexte d'annotation Nullable aux paramètres du projet.
- `#nullable disable warnings` : Définit le contexte d'avertissement Nullable sur *désactivé*.
- `#nullable enable warnings` : Définit le contexte d'avertissement Nullable sur *activé*.
- `#nullable restore warnings` : Restaure le contexte d'avertissement Nullable aux paramètres du projet.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#region (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#region` vous permet de spécifier un bloc de code que vous pouvez développer ou réduire lors de l'utilisation de la fonctionnalité mode plan de l'éditeur de code. Dans les fichiers de code volumineux, il peut être pratique de réduire ou masquer une ou plusieurs régions pour vous concentrer sur la partie du fichier sur laquelle vous êtes en train de travailler. L'exemple suivant montre comment définir une région :

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Notes

Un bloc `#region` doit se terminer par une directive #endregion.

Un bloc `#region` ne peut pas chevaucher un bloc #if. Toutefois, un bloc `#region` peut être imbriqué dans un bloc `#if` et, inversement, un bloc `#if` peut être imbriqué dans un bloc `#region`.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#endregion (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

`#endregion` marque la fin d'un bloc #region. Par exemple :

```
#region MyClass definition
class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

#pragma (référence C#)

02/11/2020 • 2 minutes to read • Edit Online

La directive `#pragma` fournit au compilateur des instructions spéciales pour la compilation du fichier dans lequel elle apparaît. Les instructions doivent être prises en charge par le compilateur. En d'autres termes, vous ne pouvez pas utiliser `#pragma` pour créer des instructions de prétraitement personnalisées. Le compilateur Microsoft C# prend en charge les deux instructions `#pragma` suivantes :

AVERTISSEMENT #pragma

somme de contrôle #pragma

Syntaxe

```
#pragma pragma-name pragma-arguments
```

Paramètres

`pragma-name`
Nom d'une directive pragma reconnue.

`pragma-arguments`
Arguments propres à la directive pragma.

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#
- AVERTISSEMENT #pragma
- somme de contrôle #pragma

#pragma warning (référence C#)

16/12/2020 • 2 minutes to read • Edit Online

`#pragma warning` peut activer ou désactiver certains avertissements.

Syntaxe

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Paramètres

`warning-list`
Liste de numéros d'avertissement séparés par des virgules. Le préfixe « CS » est facultatif.

Quand aucun numéro d'avertissement n'est spécifié, `disable` désactive tous les avertissements et `restore` active tous les avertissements.

> **NOTE**
> Pour trouver les numéros d'avertissement dans Visual Studio, générez votre projet, puis recherchez les numéros d'avertissement dans la fenêtre **Sortie**.

Exemple

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#

- Directives de préprocesseur C#
- Erreurs du compilateur C#

#pragma checksum (Référence C#)

02/11/2020 • 2 minutes to read • Edit Online

Génère des sommes de contrôle pour les fichiers sources afin de faciliter le débogage des pages ASP.NET.

Syntaxe

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Paramètres

`"filename"`
Nom du fichier dont les modifications ou les mises à jour doivent faire l'objet d'une surveillance.

`"{guid}"`
Identificateur global unique (GUID) du fichier pour l'algorithme de hachage.

`"checksum_bytes"`
Chaîne de chiffres hexadécimaux représentant les octets de la somme de contrôle. Doit être un nombre pair de chiffres hexadécimaux. S'il y a un nombre impair de chiffres, un avertissement est généré au moment de la compilation et la directive est ignorée.

Notes

Le débogueur Visual Studio utilise une somme de contrôle pour s'assurer de toujours trouver la bonne source. Le compilateur calcule la somme de contrôle pour un fichier source, puis envoie la sortie vers le fichier de base de données du programme (PDB). Le débogueur utilise ensuite le fichier PDB à comparer avec la somme de contrôle qu'il calcule pour le fichier source.

Cette solution ne fonctionne pas pour les projets ASP.NET, car la somme de contrôle est calculée pour le fichier source généré, au lieu du fichier .aspx. Pour résoudre ce problème, `#pragma checksum` fournit une prise en charge de la somme de contrôle pour les pages ASP.NET.

Quand vous créez un projet ASP.NET dans Visual C#, le fichier source généré contient une somme de contrôle pour le fichier .aspx, à partir duquel la source est générée. Le compilateur écrit ensuite ces informations dans le fichier PDB.

Si le compilateur ne rencontre aucune directive `#pragma checksum` dans le fichier, il calcule la somme de contrôle et écrit la valeur dans le fichier PDB.

Exemple

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Directives de préprocesseur C#

Options du compilateur C#

02/11/2020 • 2 minutes to read • Edit Online

Le compilateur produit des fichiers exécutables (.exe), des bibliothèques de liens dynamiques (.dll) ou des modules de code (.netmodule).

Chaque option du compilateur est disponible sous deux formes : **-option** et **/option**. La documentation présente uniquement la forme **-option**.

Dans Visual Studio, vous définissez les options du compilateur dans le fichier *web.config* . Pour plus d'informations, consultez <compiler> élément.

Contenu de cette section

- Génération à partir de la ligne de commande avec csc.exe Informations sur la création d'une application Visual C# à partir de la ligne de commande.
- Comment définir des variables d'environnement pour la ligne de commande Visual Studio Fournit la procédure d'exécution de *VsDevCmd.bat* pour activer des générations à partir de la ligne de commande.
- Options du compilateur C# classées par catégorie Liste catégorique des options du compilateur.
- Options du compilateur C# par ordre alphabétique Liste alphabétique des options du compilateur.

Sections connexes

- Générer, page du concepteur de projets Définition des propriétés qui régissent la compilation, la génération et le débogage de votre projet. Contient des informations sur les étapes de génération personnalisée dans les projets Visual C#.
- Builds par défaut et personnalisées Informations sur les types de build et les configurations.
- Préparation et gestion des builds Procédures de génération dans l'environnement de développement Visual Studio.

Génération à partir de la ligne de commande avec csc.exe

16/12/2020 • 7 minutes to read • Edit Online

Vous pouvez appeler le compilateur C# en tapant le nom de son fichier exécutable (*csc.exe*) à l'invite de commandes.

Si vous utilisez la fenêtre **Invite de commandes développeur pour Visual Studio**, toutes les variables d'environnement nécessaires sont définies automatiquement. Pour plus d'informations sur la façon d'accéder à cet outil, consultez la rubrique Invite de commandes développeur pour Visual Studio.

Si vous utilisez une fenêtre d'invite de commandes standard, vous devez ajuster votre chemin d'accès avant de pouvoir appeler *csc.exe* à partir de n'importe quel sous-répertoire de votre ordinateur. Vous devez également exécuter *VsDevCmd.bat* pour définir les variables d'environnement appropriées pour prendre en charge les générations à partir de la ligne de commande. Pour plus d'informations sur *VsDevCmd.bat*, y compris des instructions sur la façon de le Rechercher et de l'exécuter, consultez comment définir des variables d'environnement pour la ligne de commande Visual Studio.

Si vous utilisez un ordinateur disposant uniquement du Kit de développement logiciel (SDK) Windows, vous pouvez utiliser le compilateur C# dans l'**invite de commandes du kit SDK**, disponible à partir de l'option de menu **Microsoft .NET Framework SDK**.

Vous pouvez également utiliser MSBuild pour générer des programmes en C# par programmation. Pour plus d'informations, consultez MSBuild.

Le fichier exécutable *csc.exe* se trouve généralement dans le dossier Microsoft. NET\Framework \ *<Version>* sous le répertoire *Windows* . Son emplacement peut varier en fonction de la configuration exacte de l'ordinateur utilisé. Si plusieurs versions du .NET Framework sont installées sur votre ordinateur, vous trouverez plusieurs versions de ce fichier. Pour plus d'informations sur ces installations, consultez Guide pratique pour déterminer les versions installées du .NET Framework.

> **TIP**
>
> Quand vous générez un projet à l'aide de l'IDE de Visual Studio, vous pouvez introduire la commande **csc** et ses options de compilation associées dans la fenêtre **Sortie**. Pour afficher ces informations, suivez les instructions figurant dans Comment : afficher, enregistrer et configurer des fichiers journaux de génération pour définir le niveau de détail des données de journal sur **Normal** ou **Détaillé**. Après avoir régénéré votre projet, recherchez **csc** dans la fenêtre **Sortie** pour rechercher l'appel du compilateur C#.

Dans cette rubrique

- Règles de syntaxe de ligne de commande
- Exemples de lignes de commande
- Différences entre le compilateur C# et la sortie du compilateur C++

Règles de syntaxe de ligne de commande pour le compilateur C#

Le compilateur C# utilise les règles suivantes quand il interprète les arguments spécifiés dans la ligne de commande du système d'exploitation :

- Les arguments sont délimités par un espace blanc, qui peut être un espace ou une tabulation.
- Le signe insertion (^) n'est pas reconnu comme caractère d'échappement ni comme délimiteur. Ce caractère est traité par l'analyseur de ligne de commande du système d'exploitation avant d'être passé au tableau `argv` du programme.
- Une chaîne placée entre guillemets doubles ("chaîne") est interprétée comme un argument unique, quels que soient les espaces blancs inclus. Une chaîne entre guillemets peut être incorporée dans un argument.
- Un guillemet double précédé d'une barre oblique inverse (\") est interprété comme un caractère guillemet double littéral (").
- Les barres obliques inverses sont interprétées littéralement, sauf si elles précèdent immédiatement un guillemet double.
- Si un nombre pair de barres obliques inverses est suivi d'un guillemet double, une barre oblique inverse est placée dans le tableau `argv` pour chaque paire de barres obliques inverses, et le guillemet double est interprété comme un délimiteur de chaîne.
- Si un nombre impair de barres obliques inverses est suivi d'un guillemet double, une barre oblique inverse est placée dans le tableau `argv` pour chaque paire de barres obliques inverses, et le guillemet double est « ignoré » en raison de la barre oblique inverse restante. Cela entraîne l'ajout d'un guillemet double littéral (") dans le tableau `argv`.

Exemples de lignes de commande pour le compilateur C#

- Compile *file.cs* produisant des *File.exe*:

```
csc File.cs
```

- Compile *file.cs* produisant des *File.dll*:

```
csc -target:library File.cs
```

- Compile *file.cs* et crée *My.exe*:

```
csc -out:My.exe File.cs
```

- Compile tous les fichiers C# du répertoire actif avec les optimisations activées et définit le symbole DEBUG. La sortie est *File2.exe*:

```
csc -define:DEBUG -optimize -out:File2.exe *.cs
```

- Compile tous les fichiers C# du répertoire actif qui génèrent une version de débogage de *File2.dll*. Aucun logo ni aucun avertissement n'est affiché :

```
csc -target:library -out:File2.dll -warn:0 -nologo -debug *.cs
```

- Compile tous les fichiers C# du répertoire actif dans un fichier *.xyz* (une dll) :

```
csc -target:library -out:Something.xyz *.cs
```

Différences entre les résultats de la compilation en C# et ceux de la compilation en C++

Aucun fichier objet (*.obj*) n'est créé suite à l'appel du compilateur C#. les fichiers de sortie sont créés directement. Le compilateur C# n'a donc pas besoin d'un éditeur de liens.

Voir aussi

- Options du compilateur C#
- Options du compilateur C# par ordre alphabétique
- Options du compilateur C# par catégorie
- Arguments main () et Command-Line
- Arguments de ligne de commande
- Comment afficher les arguments de ligne de commande
- Valeurs de retour Main()

Comment : définir des variables d'environnement pour la ligne de commande Visual Studio

16/12/2020 • 2 minutes to read • Edit Online

Le fichier VsDevCmd.bat définit les variables d'environnement appropriées pour les générations à partir de la ligne de commande.

> **NOTE**
>
> Visual Studio 2015 et versions antérieures utilisaient VSVARS32.bat, pas VsDevCmd.bat dans le même but. Ce fichier était stocké dans \Program Files\Microsoft Visual Studio*Version*\Common7\Tools ou Program Files (x86)\Microsoft Visual Studio*Version*\Common7\Tools.

Si la version actuelle de Visual Studio est installée sur un ordinateur qui exécute également une version antérieure de Visual Studio, vous ne devez pas exécuter VsDevCmd.bat et VSVARS32.BAT à partir de différentes versions dans la même fenêtre d'invite de commandes. Exécutez plutôt la commande pour chaque version dans sa propre fenêtre.

Pour exécuter VsDevCmd.BAT

1. Dans le menu **Démarrer** , ouvrez le **Invite de commandes développeur pour vs 2019**. Il se trouve dans le dossier **Visual Studio 2019** .
2. Accédez au sous-répertoire \Program Files\Microsoft Visual Studio \ *version* \ *offer*\Common7\Tools ou \Program Files (x86) \Microsoft Visual Studio \ *version* \ *offer*\Common7\Tools de votre installation. (La *version* est *2019* pour la version actuelle. *Offre* correspond à *Enterprise*, *Professional* ou *Community*.)
3. Exécutez VsDevCmd.bat en tapant **VsDevCmd**.

   > **Caution**
   >
   > VsDevCmd.bat peut varier d'un ordinateur à l'autre. Ne remplacez pas un fichier VsDevCmd.bat manquant ou endommagé par un fichier VsDevCmd.bat d'un autre ordinateur. À la place, réexécutez le programme d'installation pour remplacer le fichier manquant.

Options disponibles pour VsDevCmd.BAT

Pour afficher les options disponibles pour VsDevCmd.BAT, exécutez la commande avec l'option `-help` :

```
VsDevCmd.bat -help
```

Voir aussi

- Génération à partir de la ligne de commande avec csc.exe

Options du compilateur C# par catégorie

02/11/2020 • 7 minutes to read • Edit Online

Les options du compilateur suivantes sont triées par catégorie. Pour en obtenir la liste alphabétique, consultez Options du compilateur C# par ordre alphabétique.

Optimization

OPTION	OBJECTIF
-filealign	Spécifie la taille des sections dans le fichier de sortie.
-optimiser	Active/désactive les optimisations.

Fichiers de sortie

OPTION	OBJECTIF
-deterministic	Indique au compilateur de générer un assembly dont le contenu binaire est identique dans les compilations si les entrées sont identiques.
-doc	Spécifie un fichier XML dans lequel les commentaires de documentation traités doivent être écrits.
-out	Spécifie le fichier de sortie.
-pathmap	Spécifie un mappage pour la sortie de noms de chemin d'accès source par le compilateur.
-pdb	Spécifie le nom et l'emplacement du fichier .pdb.
-plateforme	Spécifie la plateforme de sortie.
-preferreduilang	Spécifie un langage pour les résultats de la compilation.
-refout	Génère un assembly de référence en plus de l'assembly principal.
-refonly	Génère un assembly de référence au lieu d'un assembly principal.
-cible	Spécifie le format du fichier de sortie à l'aide de l'une des options suivantes : -target : appcontainerexe, -target : exe, -target : Library, -target : module, -target : winexeou -target : winmdobj.
NomModule<string>	Spécifiez le nom du module source.

Assemblys .NET

OPTION	OBJECTIF
-addmodule	Spécifie un ou plusieurs modules à inclure dans cet assembly.
-delaysign	Indique au compilateur d'ajouter la clé publique mais de laisser l'assembly non signé.
-keycontainer	Spécifie le nom du conteneur de la clé de chiffrement.
-keyfile	Spécifie le nom du fichier contenant la clé de chiffrement.
-lib	Spécifie l'emplacement des assemblys référencés par le biais de -reference.
-nostdlib	Indique au compilateur de ne pas importer la bibliothèque standard (mscorlib.dll).
-publicsign	Applique une clé publique sans signature de l'assembly, mais définit le bit dans l'assembly pour indiquer que l'assembly est signé.
-Référence	Importe des métadonnées à partir d'un fichier qui contient un assembly.
-analyzer	Exécutez les analyseurs à partir de cet assembly (forme abrégée : /a).
-additionalfile	Nomme des fichiers supplémentaires qui n'affectent pas directement la génération de code, mais peuvent être utilisés par des analyseurs pour produire des erreurs ou des avertissements.
-embed	Incorporer tous les fichiers sources du PDB.
Corporation<file list>	Incorporer des fichiers spécifiques dans le PDB.

Débogage/vérification des erreurs

OPTION	OBJECTIF
-bugreport	Crée un fichier qui contient des informations qui facilitent le signalement d'un bogue.
-activé	Indique si l'arithmétique sur les entiers qui dépasse les limites du type de données entraîne une exception au moment de l'exécution.
-débogage	Indique au compilateur d'émettre des informations de débogage.
-errorreport	Définit le comportement de signalement d'erreurs.

OPTION	OBJECTIF
-fullpaths	Spécifie le chemin d'accès absolu au fichier de sortie du compilateur.
-nowarn	Supprime la génération par le compilateur des avertissements spécifiés.
-nullable	Spécifie l'option de contexte Nullable.
-warn	Définit le niveau d'avertissement.
-warnaserror	Transforme les avertissements en erreurs.
RuleSet<file>	Spécifiez un fichier ruleset qui désactive des diagnostics spécifiques.

Préprocesseur

OPTION	OBJECTIF
-définir	Définit les symboles du préprocesseur.

Ressources

OPTION	OBJECTIF
-lien	Rend les informations de type COM dans les assemblys spécifiés disponibles pour le projet.
-linkresource	Crée un lien à une ressource managée.
-ressource	Incorpore une ressource .NET dans le fichier de sortie.
-win32icon	Spécifie un fichier .ico à insérer dans le fichier de sortie.
-win32res	Spécifie une ressource Win32 à insérer dans le fichier de sortie.

Divers

OPTION	OBJECTIF
@	Spécifie un fichier réponse.
-?	Répertorie les options du compilateur dans stdout.
-baseaddress	Spécifie l'adresse de base préférée à laquelle charger une DLL.
-codepage	Spécifie la page de codes à utiliser pour tous les fichiers de code source inclus dans la compilation.

OPTION	OBJECTIF
-aide	Répertorie les options du compilateur dans stdout.
-highentropyva	Spécifie que le fichier exécutable prend en charge la randomisation du format d'espace d'adresse (ASLR).
-langversion	Spécifie la version de langage : Default, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 ou Latest
-main	Spécifie l'emplacement de la méthode **Main**.
-noconfig	Indique au compilateur ne pas compiler avec csc.rsp.
-nologo	Supprime les informations de bannière du compilateur.
-recurse	Recherche des fichiers sources à compiler dans les sous-répertoires.
-subsystemversion	Spécifie la version minimale du sous-système utilisable par le fichier exécutable.
-unsafe	Active la compilation du code qui utilise le mot clé unsafe.
-utf8output	Affiche les résultats de la compilation au format d'encodage UTF-8.
-parallel[+|-]	Indique s'il faut utiliser la build simultanée (+).
-checksumalgorithm:<alg>	Spécifiez l'algorithme de calcul de la somme de contrôle du fichier source stockée dans le fichier PDB. Les valeurs prises en charge sont : SHA256 (valeur par défaut) ou SHA1. En raison de problèmes de collision avec SHA-1, Microsoft recommande SHA-256.

Options obsolètes

OPTION	OBJECTIF
-incremental	Active la compilation incrémentielle.

Voir aussi

- Options du compilateur C#
- Options du compilateur C# par ordre alphabétique
- Comment : définir des variables d'environnement pour la ligne de commande Visual Studio

Options du compilateur C# par ordre alphabétique

02/11/2020 • 7 minutes to read • Edit Online

Les options du compilateur suivantes sont triées par ordre alphabétique. Pour obtenir la liste par catégorie, consultez Options du compilateur C# par catégorie.

OPTION	OBJECTIF
@	Lit un fichier réponse pour obtenir plus d'options.
-?	Affiche un message d'utilisation dans stdout.
-additionalfile	Nomme des fichiers supplémentaires qui n'affectent pas directement la génération de code, mais peuvent être utilisés par des analyseurs pour produire des erreurs ou des avertissements.
-addmodule	Lie les modules spécifiés dans cet assembly.
-analyzer	Exécute les analyseurs à partir de cet assembly (forme abrégée : -a).
-appconfig	Spécifie l'emplacement du fichier app.config au moment de la liaison d'assembly.
-baseaddress	Spécifie l'adresse de base de la bibliothèque à générer.
-bugreport	Crée un fichier de rapport de bogue. Ce fichier est envoyé avec les informations d'incident s'il est utilisé avec -errorprompt:prompt ou -errorreport:send.
-activé	Indique au compilateur de générer des contrôles de dépassement de capacité.
-checksumalgorithm:<alg>	Spécifie l'algorithme de calcul de la somme de contrôle du fichier source stockée dans le fichier PDB. Les valeurs prises en charge sont : SHA256 (valeur par défaut) ou SHA1. En raison de problèmes de collision avec SHA-1, Microsoft recommande SHA-256.
-codepage	Spécifie la page de codes à utiliser à l'ouverture des fichiers sources.
-débogage	Émet des informations de débogage.
-define	Définit des symboles de compilation conditionnelle.
-delaysign	Diffère la signature de l'assembly en utilisant uniquement la partie publique de la clé de nom fort.

OPTION	OBJECTIF
-deterministic	Indique au compilateur de générer un assembly dont le contenu binaire est identique dans les compilations si les entrées sont identiques.
-doc	Spécifie un fichier de documentation XML à générer.
-embed	Incorporer tous les fichiers sources du PDB.
Corporation<file list>	Incorporer des fichiers spécifiques dans le PDB.
-errorendlocation	Ligne et colonne de sortie de l'emplacement de fin de chaque erreur.
Journal<file>	Spécifier un fichier pour journaliser tous les diagnostics du compilateur et de l'analyseur.
-errorreport	Spécifie comment gérer les erreurs internes du compilateur : invite, envoi et aucune gestion. La valeur par défaut est aucune gestion.
-filealign	Spécifie l'alignement utilisé pour les sections du fichier de sortie.
-fullpaths	Indique au compilateur de générer des chemins d'accès complets.
-aide	Affiche un message d'utilisation dans stdout.
-highentropyva	Spécifie que la randomisation du format d'espace d'adresse (ASLR) d'entropie élevée est prise en charge.
-incremental	Active la compilation incrémentielle [obsolète].
-keycontainer	Spécifie un conteneur de clé de nom fort.
-keyfile	Spécifie un fichier de clé de nom fort.
langversion<string>	Spécifie la version de langage : Default, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 ou Latest
-lib	Spécifie les répertoires supplémentaires dans lesquels rechercher des références.
-lien	Rend les informations de type COM dans les assemblys spécifiés disponibles pour le projet.
-linkresource	Lie la ressource spécifiée à cet assembly.
-main	Spécifie le type qui contient le point d'entrée (ignorez tous les autres points d'entrée possibles)
-moduleassemblyname	Spécifie un assembly dont les types non publics sont accessibles par un .netmodule.

OPTION	OBJECTIF
NomModule<string>	Spécifiez le nom du module source.
-noconfig	Indique au compilateur ne pas inclure automatiquement de fichier CSC.RSP.
-nologo	Supprime le message de copyright du compilateur.
-nostdlib	Indique au compilateur de ne pas référencer la bibliothèque standard (mscorlib.dll).
-nowarn	Désactive des messages d'avertissement spécifiques.
-nowin32manifest	Indique au compilateur de ne pas incorporer un manifeste d'application dans le fichier exécutable.
-nullable	Spécifie l'option de contexte Nullable.
-optimiser	Active/désactive les optimisations.
-out	Spécifie le nom de fichier de sortie (par défaut : nom de base du fichier avec classe principale ou premier fichier).
-parallel[+|-]	Indique s'il faut utiliser la build simultanée (+).
-pathmap	Spécifie un mappage pour la sortie de noms de chemin d'accès source par le compilateur.
-pdb	Spécifie le nom et l'emplacement du fichier .pdb.
-plateforme	Limite les plateformes sur lesquelles ce code peut s'exécuter : x86, Itanium, x64, anycpu ou anycpu32bitpreferred. La valeur par défaut est anycpu.
-preferreduilang	Spécifie la langue à utiliser pour la sortie du compilateur.
-publicsign	Applique une clé publique sans signature de l'assembly, mais définit le bit dans l'assembly pour indiquer que l'assembly est signé.
-recurse	Inclut tous les fichiers dans le répertoire et les sous-répertoires actuels en fonction des spécifications de caractères génériques.
-Référence	Référence les métadonnées à partir des fichiers d'assembly spécifiés.
-refout	Génère un assembly de référence en plus de l'assembly principal.
-refonly	Génère un assembly de référence au lieu d'un assembly principal.

OPTION	OBJECTIF
-reportanalyzer	Créer un rapport sur les informations supplémentaires de l'analyseur, comme l'heure d'exécution.
-ressource	Incorpore la ressource spécifiée.
RuleSet<file>	Spécifiez un fichier ruleset qui désactive des diagnostics spécifiques.
-subsystemversion	Spécifie la version minimale du sous-système utilisable par le fichier exécutable.
-cible	Spécifie le format du fichier de sortie à l'aide de l'une des options suivantes : -target : appcontainerexe, -target : exe, -target : Library, -target : module, -target : winexe, -target : winmdobj.
-unsafe	Autorise le code unsafe.
-utf8output	Génère des messages du compilateur encodés en UTF-8.
-version	Afficher le numéro de version du compilateur et quitter.
-warn	Définit le niveau d'avertissement (entre 0 et 4).
-warnaserror	Signale des avertissements spécifiques en tant qu'erreurs.
-win32icon	Utilise cette icône pour la sortie.
-win32manifest	Spécifie un fichier manifeste win32 personnalisé.
-win32res	Spécifie le fichier de ressources win32 (.res).

Voir aussi

- Options du compilateur C#
- Options du compilateur C# par catégorie
- Comment : définir des variables d'environnement pour la ligne de commande Visual Studio
- <compiler> Appartient

@ (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

L'option @ vous permet de spécifier un fichier qui contient les options du compilateur et les fichiers de code source à compiler.

Syntaxe

```
@response_file
```

Arguments

`response_file`
Fichier qui répertorie des options du compilateur ou des fichiers de code source à compiler.

Notes

Les options du compilateur et les fichiers de code source seront traités par le compilateur exactement comme s'ils avaient été spécifiés sur la ligne de commande.

Pour spécifier plusieurs fichiers réponse dans une compilation, spécifiez plusieurs options de fichier réponse. Par exemple :

```
@file1.rsp @file2.rsp
```

Dans un fichier réponse, plusieurs options du compilateur et fichiers de code source peuvent apparaître sur une même ligne. En revanche, si vous spécifiez une seule option du compilateur, celle-ci doit obligatoirement figurer sur une seule et même ligne. Les fichiers réponse peuvent contenir des commentaires, précédés du symbole #.

La spécification d'options du compilateur à partir d'un fichier réponse est identique à la spécification de ces commandes sur la ligne de commande. Pour plus d'informations, consultez Génération à partir de la ligne de commande.

Le compilateur traite les options de commande au fur et à mesure qu'il les rencontre. Par conséquent, des arguments de ligne de commande peuvent substituer des options précédemment affichées dans des fichiers réponse. Inversement, les options d'un fichier réponse substituent les options affichées précédemment dans la ligne de commande ou dans d'autres fichiers réponse.

C# fournit le fichier csc.rsp, qui se trouve dans le même répertoire que le fichier csc.exe. Pour plus d'informations sur CSC. rsp, consultez -noconfig .

Cette option du compilateur ne peut pas être définie dans l'environnement de développement Visual Studio, ni être modifiée par programmation.

Exemple

Voici quelques lignes extraites d'un exemple de fichier réponse :

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

Voir aussi

- Options du compilateur C#

-addmodule (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette option ajoute un module créé avec le commutateur target:module à la compilation actuelle.

Syntaxe

```
-addmodule:file[;file2]
```

Arguments

`file`, `file2`
Fichier de sortie qui contient des métadonnées. Le fichier ne peut pas contenir un manifeste d'assembly. Pour importer plusieurs fichiers, séparez les noms des fichiers par une virgule ou un point-virgule.

Notes

Tous les modules ajoutés par **-addmodule** doivent se trouver dans le même répertoire que le fichier de sortie au moment de l'exécution. Vous pouvez donc spécifier un module dans n'importe quel répertoire au moment de la compilation, mais le module doit se trouver dans le répertoire de l'application au moment de l'exécution. Si le module n'est pas dans le répertoire de l'application au moment de l'exécution, vous obtiendrez TypeLoadException.

`file` ne peut pas contenir un assembly. Si, par exemple, le fichier de sortie a été créé avec -target:module, ses métadonnées peuvent être importées avec **-addmodule**.

Si le fichier de sortie a été créé avec une option **-target** autre que **-target:module**, ses métadonnées ne peuvent pas être importées avec **-addmodule**, mais peuvent l'être avec -reference.

Cette option du compilateur n'est pas disponible dans Visual Studio ; un projet ne peut pas référencer un module. Par ailleurs, cette option du compilateur ne peut pas être changée par programmation.

Exemple

Compilez le fichier source `input.cs` et ajoutez les métadonnées de `metad1.netmodule` et `metad2.netmodule` pour produire `out.exe` :

```
csc -addmodule:metad1.netmodule;metad2.netmodule -out:out.exe input.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions
- Assemblys multifichiers
- Comment : générer un assembly multifichier

-appconfig (Options du compilateur C#)

02/11/2020 • 3 minutes to read • Edit Online

L'option du compilateur **-appconfig** permet à une application C# d'indiquer l'emplacement du fichier de configuration de l'application (app.config) d'un assembly au CLR (Common Language Runtime) au moment de la liaison de l'assembly.

Syntaxe

```
-appconfig:file
```

Arguments

`file`
Obligatoire. Fichier de configuration de l'application qui contient des paramètres de liaison de l'assembly.

Notes

-appconfig est notamment utilisé dans des scénarios avancés dans lesquels un assembly doit simultanément référencer à la fois la version du .NET Framework et la version du .NET Framework pour Silverlight d'un assembly de référence particulier. Par exemple, un concepteur XAML écrit dans WPF (Windows Presentation Foundation) devra peut-être référencer à la fois le bureau WPF pour l'interface utilisateur du concepteur et le sous-ensemble de WPF qui est fourni avec Silverlight. Le même assembly de concepteur doit accéder aux deux assemblys. Par défaut, les références séparées provoquent une erreur du compilateur parce que la liaison d'assembly considère les deux assemblys comme équivalents.

L'option du compilateur **-appconfig** vous permet de spécifier l'emplacement d'un fichier app.config qui désactive le comportement par défaut à l'aide d'une balise `<supportPortability>`, comme indiqué dans l'exemple suivant.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

Le compilateur passe l'emplacement du fichier à la logique de liaison d'assembly du CLR.

> **NOTE**
> Si vous utilisez MSBuild (Microsoft Build Engine) pour générer votre application, vous pouvez définir l'option du compilateur **-appconfig** en ajoutant une balise de propriété au fichier .csproj. Pour utiliser le fichier app.config qui est déjà défini dans le projet, ajoutez la balise de propriété `<UseAppConfigForCompiler>` au fichier .csproj et affectez-lui la valeur `true`. Pour spécifier un autre fichier app.config, ajoutez la balise de propriété `<AppConfigForCompiler>` et définissez sa valeur sur l'emplacement du fichier.

Exemple

L'exemple suivant affiche un fichier app.config qui permet à une application d'avoir des références à la fois à l'implémentation .NET Framework et à l'implémentation .NET Framework pour Silverlight de tout assembly .NET Framework qui existe dans les deux implémentations. L'option du compilateur **-appconfig** spécifie l'emplacement de ce fichier app.config.

```
<configuration>
      <runtime>
      <assemblyBinding>
            <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
            <supportPortability PKT="31bf3856ad364e35" enable="false"/>
      </assemblyBinding>
      </runtime>
</configuration>
```

Voir aussi

- <supportPortability> Appartient
- Options du compilateur C# par ordre alphabétique

-baseaddress (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-baseaddress** vous permet de spécifier l'adresse de base préférée à laquelle doit être chargée une DLL. Pour plus d'informations sur le moment et la raison de l'utilisation de cette option, consultez Larry Osterman's WebLog.

Syntaxe

```
-baseaddress:address
```

Arguments

`address`

Adresse de base de la DLL. Cette adresse peut être spécifiée sous forme d'un nombre décimal, hexadécimal ou octal.

Notes

L'adresse de base par défaut d'une DLL est définie par le common language runtime .NET.

Sachez que le dernier chiffre de cette adresse sera arrondi. Par exemple, si vous spécifiez l'adresse 0x11110001, elle est arrondie à 0x11110000.

Pour terminer le processus de signature d'une DLL, utilisez SN.EXE avec l'option -R.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Cliquez sur le bouton **Avancé** .
4. Modifiez la propriété **Adresse de base de la DLL**.

   Pour définir cette option du compilateur par programmation, consultez BaseAddress.

Voir aussi

- ProcessModule.BaseAddress
- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-bugreport (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Spécifie que les informations de débogage doivent être placées dans un fichier pour une future analyse.

Syntaxe

```
-bugreport:file
```

Arguments

`file`
Nom du fichier qui doit contenir votre rapport de bogues.

Notes

L'option **-bugreport** spécifie que les informations suivantes doivent être placées dans `file` :

- Une copie de tous les fichiers de code source de la compilation.
- Une liste des options du compilateur utilisées dans la compilation.
- Les informations de version concernant votre compilateur, votre runtime et votre système d'exploitation.
- Les assemblys et les modules référencés, enregistrés sous forme de chiffres hexadécimaux, à l'exception des assemblys qui sont fournis avec .NET et le kit de développement logiciel (SDK) .NET.
- Les résultats de la compilation, le cas échéant.
- Une description du problème, qui vous sera demandée.
- Une description de vos suggestions de résolution du problème, qui vous sera demandée.

Si cette option est utilisée avec **-errorreport:prompt** ou **-errorreport:send**, les informations figurant dans le fichier seront envoyées à Microsoft Corporation.

Étant donné qu'une copie de tous les fichiers de code source sera placée dans `file`, vous souhaiterez peut-être reproduire l'erreur de code suspecte dans le programme le plus court possible.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Notez que le contenu du fichier généré expose le code source, ce qui pourrait entraîner une divulgation d'informations non voulue.

Voir aussi

- Options du compilateur C#
- -errorReport (options du compilateur C#)
- Gestion des propriétés des projets et des solutions

-checked (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-checked** spécifie si une instruction arithmétique entière qui produit une valeur en dehors de la plage du type de données, et qui n'est pas dans la portée d'un mot clé checked ou unchecked, doit provoquer une exception au moment de l'exécution.

Syntaxe

```
-checked[+ | -]
```

Notes

Une instruction arithmétique entière qui est dans la portée d'un mot clé `checked` ou `unchecked` n'est pas soumise à l'effet de l'option **-checked**.

Si une instruction arithmétique entière qui n'est pas dans la portée d'un mot clé `checked` ou `unchecked` produit une valeur en dehors de la plage du type de données, et que **-checked+** (ou **-checked**) est utilisé dans la compilation, cette instruction provoque une exception au moment de l'exécution. Si **-checked-** est utilisé dans la compilation, cette instruction ne provoque pas d'exception au moment de l'exécution.

La valeur par défaut de cette option est **-checked-**. La vérification du dépassement de capacité est désactivée.

Certains outils automatisés qui sont utilisés pour générer des applications volumineuses définissent l'option -checked sur +. Un scénario d'usage de l'option -checked- est de substituer l'option globale par défaut de l'outil en spécifiant -checked-.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet. Pour plus d'informations, consultez Générer, page du Concepteur de projets (C#).
2. Cliquez sur la page de propriétés **Générer**.
3. Cliquez sur le bouton **Avancé** .
4. Modifiez la propriété **Vérifier la débordement arithmétique** .

Pour accéder à cette option du compilateur par programmation, consultez CheckForOverflowUnderflow.

Exemple

La commande suivante compile `t2.cs`. L'utilisation de `-checked` dans la commande spécifie que toute instruction arithmétique entière dans le fichier qui n'est pas dans la portée d'un mot clé `checked` ou `unchecked`, et qui produit une valeur en dehors de la plage du type de données, provoque une exception au moment de l'exécution.

```
csc t2.cs -checked
```

Voir aussi

- Options du compilateur C#

- Gestion des propriétés des projets et des solutions

-codepage (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Cette option spécifie la page de codes à utiliser pendant la compilation si la page exigée n'est pas la page de codes par défaut actuelle du système.

Syntaxe

```
-codepage:id
```

Arguments

`id`

Identificateur de la page de codes à utiliser pour tous les fichiers de code source de la compilation.

Notes

Le compilateur tente tout d'abord d'interpréter tous les fichiers sources comme des fichiers UTF-8. Si vos fichiers de code source sont encodés dans un format autre que UTF-8 et s'ils utilisent des caractères autres que des caractères ASCII 7 bits, utilisez l'option **-codepage** pour spécifier la page de codes à utiliser. **-codepage** s'applique à tous les fichiers de code source inclus dans votre compilation.

Pour plus d'informations sur la façon de savoir quelles pages de codes sont prises en charge sur votre système, consultez GetCPInfo.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-debug (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-debug** fait en sorte que le compilateur génère des informations de débogage et les place dans les fichiers de sortie.

Syntaxe

```
-debug[+ | -]
-debug:{full | pdbonly}
```

Arguments

`+` | `-`

Si vous spécifiez `+` ou simplement **-debug**, le compilateur génère des informations de débogage et les place dans un fichier de base de données du programme (.pdb). Si vous spécifiez `-`, qui est en vigueur si vous ne spécifiez pas **-debug**, aucune information de débogage n'est créée.

`full` | `pdbonly`

Indique le type d'informations de débogage générées par le compilateur. L'argument complet, qui est en vigueur si vous ne spécifiez pas **-debug:pdbonly**, permet d'attacher un débogueur au programme en cours d'exécution. Le fait de spécifier pdbonly active le débogage du code source quand le programme est démarré dans le débogueur, mais affiche uniquement un assembleur quand le programme en cours d'exécution est attaché au débogueur.

Notes

Utilisez cette option pour créer des versions Debug. Si **-debug**, **-debug+** ou **-debug:full** n'est pas spécifié, vous ne pourrez pas déboguer le fichier de sortie de votre programme.

Si vous utilisez **-debug:full**, sachez qu'il y a un impact sur la vitesse et la taille du code optimisé JIT, ainsi qu'un faible impact sur la qualité du code avec **-debug:full**. Nous vous recommandons d'utiliser **-debug:pdbonly** ou aucun PDB pour la génération du code de version finale.

> **NOTE**
>
> L'une des différences entre **-debug:pdbonly** et **-debug:full** est qu'avec **-debug:full**, le compilateur émet un DebuggableAttribute, qui est utilisé pour signaler au compilateur JIT que des informations de débogage sont disponibles. Ainsi, une erreur se produit si votre code contient un DebuggableAttribute avec la valeur false si vous utilisez **-debug:full**.

Pour plus d'informations sur la façon de configurer les performances de débogage d'une application, consultez Simplification du débogage d'une image.

Pour changer l'emplacement du fichier .pdb, consultez -pdb (Options du compilateur C#).

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Cliquez sur le bouton **Avancé** .

4. Modifiez la propriété **Infos de débogage**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez DebugSymbols.

Exemple

Placez les informations de débogage dans le fichier de sortie `app.pdb` :

```
csc -debug -pdb:app.pdb test.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-define (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-define** définit `name` comme symbole dans tous les fichiers de code source de votre programme.

Syntaxe

```
-define:name[;name2]
```

Arguments

`name`, `name2`
Nom de chaque symbole que vous souhaitez définir.

Notes

L'option **-define** revient à utiliser une directive de préprocesseur #define, sauf que l'option de compilateur est appliquée à tous les fichiers du projet. Un symbole reste défini dans un fichier source jusqu'à ce qu'une directive #undef dans le fichier source en supprime la définition. Quand vous utilisez l'option -define, une directive `#undef` dans un fichier n'a aucun effet sur les autres fichiers de code source du projet.

Vous pouvez utiliser les symboles créés par cette option avec #if, #else, #elif et #endif pour effectuer une compilation conditionnelle des fichiers sources.

-d est la forme abrégée de **-define**.

Pour définir plusieurs symboles avec **-define**, séparez chaque nom de symbole par un point-virgule ou une virgule. Exemple :

```
-define:DEBUG;TUESDAY
```

Le compilateur C# lui-même ne définit aucun symbole ni aucune macro que vous pouvez utiliser dans votre code source ; toutes les définitions de symbole doivent être définies par l'utilisateur.

> **NOTE**
> La directive C# `#define` ne permet de donner une valeur à un symbole, comme dans les langages tels que C++. Par exemple, l'option `#define` ne peut pas être utilisée pour créer une macro ni définir une constante. Si vous devez définir une constante, utilisez une variable `enum`. Si vous souhaitez créer une macro de style C++, envisagez l'alternative que constituent les génériques. Comme les macros sont notoirement sujettes aux erreurs, C# n'autorise pas leur utilisation, mais fournit des solutions plus sûres.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Dans l'onglet **Générer**, tapez le symbole à définir dans la zone **Symboles de compilation conditionnelle**. Par exemple, si vous utilisez l'exemple de code qui suit, tapez simplement `xx` dans la zone de texte.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez DefineConstants.

Exemple

```
// preprocessor_define.cs
// compile with: -define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-delaysign (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Cette option fait en sorte que le compilateur réserve de l'espace dans le fichier de sortie afin qu'une signature numérique puisse être ajoutée ultérieurement.

Syntaxe

```
-delaysign[ + | - ]
```

Arguments

`+` | `-`

Utilisez **-delaysign-** si vous souhaitez obtenir un assembly complètement signé. Utilisez **-delaysign+** si vous souhaitez uniquement placer la clé publique dans l'assembly. **-delaysign-** est l'option par défaut.

Notes

L'option **-delaysign** n'a aucun effet, sauf si elle est utilisée avec -keyfile ou -keycontainer.

Les options **-delaysign** et **-publicsign** s'excluent mutuellement.

Quand vous demandez un assembly totalement signé, le compilateur hache le fichier qui contient le manifeste (métadonnées de l'assembly) et signe ce hachage avec la clé privée. L'opération crée une signature numérique qui est stockée dans le fichier contenant le manifeste. Pour un assembly avec signature différée, le compilateur ne calcule pas, ni ne stocke la signature, mais réserve de l'espace dans le fichier pour que la signature puisse être ajoutée par la suite.

Par exemple, l'utilisation de **-delaysign+** permet à un testeur de placer l'assembly dans le cache global. Après un test, vous pouvez signer complètement l'assembly en plaçant la clé privée dans l'assembly à l'aide de l'utilitaire Assembly Linker.

Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Modifiez la propriété **Différer la signature uniquement**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez DelaySign.

Voir aussi

- Option -publicsign C#
- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-deterministic

02/11/2020 • 2 minutes to read • Edit Online

Indique au compilateur de générer un assembly dont la sortie octet-pour-octet est identique dans les compilations pour les entrées identiques.

Syntaxe

```
-deterministic
```

Notes

Par défaut, le résultat de la compilation d'un ensemble défini d'entrées est unique, étant donné que le compilateur ajoute un horodatage et un GUID qui est généré à partir de nombres aléatoires. Utilisez l'option `-deterministic` pour générer un *assembly déterministe*, dont le contenu binaire sera identique dans les compilations tant que l'entrée restera la même.

Le compilateur considère les entrées suivantes à des fins déterministes :

- La séquence des paramètres de ligne de commande.
- Le contenu du fichier réponse .rsp du compilateur.
- La version précise du compilateur utilisée et ses assemblys référencés.
- Le chemin de répertoire actif.
- Le contenu binaire de tous les fichiers explicitement passés au compilateur directement ou indirectement, notamment :
  - Fichiers sources
  - Assemblys référencés
  - Modules référencés
  - Ressources
  - Fichier de clé de nom fort
  - Fichiers réponse @
  - Analyseurs
  - Ensembles de règles
  - Autres fichiers susceptibles d'être utilisés par les analyseurs
- La culture actuelle (pour le langage dans lequel les diagnostics et les messages d'exception sont produits).
- L'encodage par défaut (ou la page de codes active) si l'encodage n'est pas spécifié.
- L'existence, la non-existence et le contenu des fichiers sur les chemins de recherche du compilateur (spécifiés, par exemple, par `-lib` ou `-recurse`).
- La plateforme CLR sur laquelle est exécuté le compilateur.
- La valeur de `%LIBPATH%`, qui peut affecter le chargement des dépendances de l'analyseur.

Quand les sources sont accessibles à tous, la compilation déterministe peut servir à établir si un fichier binaire est compilé à partir d'une source approuvée. Il peut aussi être utile dans un système de génération en continu pour déterminer si les étapes de génération qui sont dépendantes des changements apportés à un binaire doivent être exécutées.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-doc (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-doc** vous permet de placer des commentaires de documentation dans un fichier XML.

Syntaxe

```
-doc:file
```

Arguments

`file`
Fichier de sortie XML, qui est rempli avec les commentaires des fichiers de code source de la compilation.

Notes

Dans les fichiers de code source, les commentaires de documentation qui précèdent les éléments suivants peuvent être traités et ajoutés au fichier XML :

- Types définis par l'utilisateur, tels qu'une classe, un délégué ou une interface
- Membres tels qu'un champ, un événement, une propriété ou une méthode

Le fichier de code source qui contient Main est sorti en premier dans le document XML.

Pour utiliser le fichier .xml généré avec la fonctionnalité IntelliSense, faites en sorte que le nom du fichier .xml soit identique à l'assembly que vous souhaitez prendre en charge, puis vérifiez que le fichier .xml est dans le même répertoire que l'assembly. Ainsi, quand l'assembly est référencé dans le projet Visual Studio, le fichier .xml est également trouvé. Pour plus d'informations, consultez Insertion de commentaires dans le code XML.

Sauf si vous compilez avec -target : module, `file` contient des <assembly> </assembly> balises qui spécifient le nom du fichier contenant le manifeste d'assembly pour le fichier de sortie de la compilation.

> **NOTE**
> L'option -doc s'applique à tous les fichiers d'entrée ou, si elle est définie dans les paramètres de projet, à tous les fichiers du projet. Pour désactiver les avertissements relatifs aux commentaires de documentation pour un fichier ou une section de code spécifique, utilisez #pragma warning.

Pour découvrir comment générer de la documentation à partir de commentaires dans votre code, consultez Balises recommandées pour les commentaires de documentation.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur l'onglet **Générer**.
3. Modifiez la propriété **Fichier de documentation XML**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez DocumentationFile.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-errorreport (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

Cette option fournit un moyen pratique de signaler une erreur interne du compilateur C# à Microsoft.

> **NOTE**
> Sur Windows Vista et Windows Server 2008, les paramètres de rapport d'erreurs que vous spécifiez pour Visual Studio ne remplacent pas les paramètres définis par l'intermédiaire de Rapport d'erreurs Windows. Les paramètres de Rapport d'erreurs Windows ont toujours priorité sur les paramètres de signalement d'erreurs de Visual Studio.

Syntaxe

```
-errorreport:{ none | prompt | queue | send }
```

Arguments

Aucune
Les rapports sur les erreurs internes du compilateur ne seront pas collectés ni envoyés à Microsoft.

invite Vous invite à envoyer un rapport lorsque vous recevez une erreur interne du compilateur. **prompt** est la valeur par défaut quand vous compilez une application dans l'environnement de développement.

file d'attente Met en file d'attente le rapport d'erreurs. Quand vous ouvrez une session avec des informations d'identification administratives, vous pouvez signaler toute défaillance qui s'est produite depuis votre dernière connexion. Vous ne serez pas invité à envoyer des rapports d'échecs plus d'une fois tous les trois jours. **queue** est la valeur par défaut quand vous compilez une application à partir de la ligne de commande.

Envoyer Envoie automatiquement des rapports d'erreurs internes du compilateur à Microsoft. Pour activer cette option, vous devez tout d'abord accepter la politique de collecte de données de Microsoft. La première fois que vous spécifiez **-errorreport:send** sur un ordinateur, un message du compilateur vous dirige vers un site web qui affiche la politique de collecte de données de Microsoft.

Notes

Une erreur interne du compilateur se produit quand le compilateur ne peut pas traiter un fichier de code source. Quand une telle erreur se produit, le compilateur ne génère pas de fichier de sortie ni de diagnostic utile que vous pouvez utiliser pour corriger votre code.

Dans les versions précédentes, quand vous receviez une erreur interne du compilateur, vous étiez invité à contacter le Support technique Microsoft pour signaler le problème. En utilisant **-errorreport**, vous pouvez fournir des informations sur l'erreur interne du compilateur à l'équipe Visual C#. Vos rapports d'erreurs peuvent aider à améliorer les versions ultérieures du compilateur.

La capacité d'un utilisateur à envoyer des rapports dépend des autorisations de stratégie ordinateur et utilisateur.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet. Pour plus d'informations, consultez Générer, page du Concepteur de projets (C#).

2. Cliquez sur la page de propriétés **Générer**.

3. Cliquez sur le bouton **Avancé** .

4. Modifiez la propriété **Rapport d'erreurs du compilateur interne**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez ErrorReport.

Voir aussi

- Options du compilateur C#

-filealign (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-filealign** permet de spécifier la taille des sections de votre fichier de sortie.

Syntaxe

```
-filealign:number
```

Arguments

`number`

Valeur qui spécifie la taille des sections dans le fichier de sortie. Les valeurs valides sont 512, 1024, 2048, 4096 et 8192. Ces valeurs sont exprimées en octets.

Notes

Chaque section est alignée sur une limite qui est un multiple de la valeur **-filealign**. Il n'existe aucune valeur fixe par défaut. Si la valeur **-filealign** n'est pas spécifiée, le Common Language Runtime choisit une valeur par défaut au moment de la compilation.

En spécifiant la taille de la section, vous affectez la taille du fichier de sortie. Il peut être utile de modifier la taille de la section pour les programmes qui sont exécutés sur des appareils de petite taille.

Utilisez DUMPBIN pour afficher des informations sur les sections de votre fichier de sortie.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Cliquez sur le bouton **Avancé** .
4. Modifiez la propriété **Alignement des fichiers**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez FileAlignment.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-fullpaths (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

Avec l'option **-fullpaths** , le compilateur spécifie le chemin d'accès complet au fichier lors de la liste des erreurs et des avertissements de compilation.

Syntaxe

```
-fullpaths
```

Notes

Par défaut, les erreurs et les avertissements générés par la compilation spécifient le nom du fichier dans lequel une erreur a été trouvée. L'option **-fullpaths** fait en sorte que le compilateur spécifie le chemin d'accès complet au fichier.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#

-help, -? (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette option envoie une liste d'options du compilateur, accompagnée d'une brève description de chaque option, à stdout.

Syntaxe

```
-help
-?
```

Notes

Si cette option est incluse dans une compilation, aucun fichier de sortie n'est créé et aucune compilation ne se produit.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-highentropyva (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option du compilateur **-highentropyva** indique au noyau Windows si un fichier exécutable particulier prend en charge la randomisation du format d'espace d'adresse (ASLR) de forte entropie.

Syntaxe

```
-highentropyva[+ | -]
```

Arguments

`+` | `-`

Cette option spécifie qu'un exécutable 64 bits ou un exécutable marqué par l'option du compilateur -platform:anycpu prend en charge un espace d'adressage virtuel d'entropie élevée. Cette option est désactivée par défaut. Utilisez **-highentropyva+** ou **-highentropyva** pour l'activer.

Notes

L'option **-highentropyva** permet à des versions compatibles du noyau Windows d'utiliser un degré d'entropie plus élevé lors de la randomisation du format de l'espace d'adresse d'un processus. En utilisant un degré d'entropie plus élevé, vous pouvez allouer un plus grand nombre d'adresses aux zones de mémoire, telles que les piles et les tas. Par conséquent, il est plus difficile de deviner l'emplacement d'une zone de mémoire.

Quand l'option du compilateur **-highentropyva** est spécifiée, l'exécutable cible et tous les modules dont il dépend doivent être capables de gérer les valeurs de pointeur supérieures à 4 Go lorsqu'ils sont exécutés en tant que processus 64 bits.

-keycontainer (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Spécifie le nom du conteneur de la clé de chiffrement.

Syntaxe

```
-keycontainer:string
```

Arguments

`string`
Nom du conteneur de clé de nom fort.

Notes

Quand l'option **-keycontainer** est utilisée, le compilateur crée un composant pouvant être partagé. Le compilateur insère une clé publique à partir du conteneur spécifié dans le manifeste d'assembly et signe l'assembly final avec la clé privée. Pour générer un fichier de clé, tapez `sn -k file` à la ligne de commande. `sn -i` installe la paire de clés dans un conteneur. Cette option n'est pas prise en charge quand le compilateur s'exécute sur CoreCLR. Pour signer un assembly en cas de génération sur CoreCLR, utilisez l'option -keyfile.

Si vous compilez avec -target:module, le nom du fichier de clé est conservé dans le module et incorporé dans l'assembly quand vous compilez ce module dans un assembly avec -addmodule.

Vous pouvez également spécifier cette option comme attribut personnalisé (System.Reflection.AssemblyKeyNameAttribute) dans le code source de n'importe quel module MSIL (Microsoft Intermediate Language).

Vous pouvez également passer vos informations de chiffrement au compilateur avec -keyfile. Utilisez -delaysign si vous voulez ajouter la clé publique au manifeste d'assembly, mais que vous voulez différer la signature de l'assembly tant qu'il n'a pas été testé.

Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Cette option de compilateur n'est pas disponible dans l'environnement de développement Visual Studio.

Vous pouvez accéder par programmation à cette option de compilateur avec AssemblyKeyContainerName.

Voir aussi

- Option -keyfile du compilateur C#
- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-keyfile (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

Spécifie le nom du fichier contenant la clé de chiffrement.

Syntaxe

```
-keyfile:file
```

Arguments

TERME	DÉFINITION
`file`	Nom du fichier contenant la clé de nom fort.

Notes

Quand cette option est utilisée, le compilateur insère la clé publique du fichier spécifié dans le manifeste d'assembly, puis signe l'assembly final avec la clé privée. Pour générer un fichier de clé, tapez sn -k `file` sur la ligne de commande.

Si vous compilez avec **-target:module**, le nom du fichier de clé est conservé dans le module et incorporé dans l'assembly créé quand vous compilez un assembly avec -addmodule.

Vous pouvez également passer vos informations de chiffrement au compilateur avec -keycontainer. Utilisez -delaysign si vous voulez obtenir un assembly partiellement signé.

Si -keyfile et -keycontainer sont spécifiés (par option de ligne de commande ou par attribut personnalisé) dans la même compilation, le compilateur essaie d'abord d'utiliser le conteneur de clé. Si cette tentative réussit, l'assembly est signé avec les informations figurant dans le conteneur de clé. Si le compilateur ne trouve pas le conteneur de clé, il essaie d'utiliser le fichier spécifié avec -keyfile. En cas de réussite, l'assembly est signé avec les informations du fichier de clé et les informations sur la clé sont installées dans le conteneur de clé (semblable à sn -i) de sorte qu'à la compilation suivante, le conteneur de clé est valide.

Notez qu'un fichier de clé peut contenir uniquement la clé publique.

Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Signature**.
3. Modifiez la propriété **Choisir un fichier de clé de nom fort**.

Vous pouvez accéder par programmation à cette option de compilateur avec AssemblyOriginatorKeyFile.

Voir aussi

- Options du compilateur C#

- Gestion des propriétés des projets et des solutions

-langversion (Options du compilateur C#)

02/11/2020 • 7 minutes to read • Edit Online

Force le compilateur à accepter uniquement la syntaxe incluse dans la spécification choisie du langage C#.

Syntaxe

```
-langversion:option
```

Arguments

`option`

Les valeurs suivantes sont valides :

VALEUR	SIGNIFICATION
`preview`	Le compilateur accepte toute la syntaxe de langage valide de la dernière préversion.
`latest`	Le compilateur accepte la syntaxe de la dernière version publiée du compilateur (versions mineures incluses).
`latestMajor` (`default`)	Le compilateur accepte la syntaxe de la dernière version principale publiée du compilateur.
`9.0`	Le compilateur accepte uniquement la syntaxe qui est incluse dans C# 9,0 ou une partie antérieure.
`8.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 8.0 ou une version antérieure.
`7.3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.3 ou une version antérieure.
`7.2`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.2 ou une version antérieure.
`7.1`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.1 ou une version antérieure.
`7`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.0 ou une version antérieure.
`6`	Le compilateur accepte uniquement la syntaxe incluse dans C# 6.0 ou une version antérieure.
`5`	Le compilateur accepte uniquement la syntaxe incluse dans C# 5.0 ou une version antérieure.

VALEUR	SIGNIFICATION
`4`	Le compilateur accepte uniquement la syntaxe incluse dans C# 4.0 ou une version antérieure.
`3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 3.0 ou une version antérieure.
`ISO-2` (ou `2`)	Le compilateur accepte uniquement la syntaxe qui est incluse dans la norme ISO/IEC 23270:2006 C# (2,0).
`ISO-1` (ou `1`)	Le compilateur accepte uniquement la syntaxe qui est incluse dans la norme ISO/IEC 23270:2003 C# (1.0/1.2).

La version du langage par défaut dépend du framework cible de votre application et de la version installée du kit SDK ou de Visual Studio. Ces règles sont définies dans l'article configuration de la version linguistique .

Remarques

Les métadonnées référencées par votre application C# ne sont pas visées par l'option de compilateur **-langversion**.

Sachant que chaque version du compilateur C# contient des extensions de la spécification du langage, **-langversion** ne vous offre pas les fonctionnalités équivalentes d'une version antérieure du compilateur.

De plus, alors que les mises à jour de la version de C# coïncident généralement avec les versions principales du .NET Framework, la nouvelle syntaxe et les nouvelles fonctionnalités ne sont pas nécessairement liées à cette version spécifique du framework. Alors que les nouvelles fonctionnalités nécessitent une nouvelle mise à jour du compilateur, publiée en même temps que la révision de C#, chaque fonctionnalité spécifique a ses propres exigences minimales relatives à l'API .NET ou au Common Language Runtime pour pouvoir s'exécuter sur des frameworks de bas niveau en incluant des packages NuGet ou d'autres bibliothèques.

Quel que soit le paramètre **-langversion** utilisé, utilisez la version actuelle du Common Language Runtime pour créer votre fichier. exe ou. dll. Les assemblys friend et -moduleassemblyname (option du compilateur C#), qui fonctionnent sous **-langversion : ISO-1**, sont une exception.

Pour d'autres façons de spécifier la version du langage C#, consultez l'article Sélectionner la version du langage c# .

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez LanguageVersion.

spécification du langage C#

VERSION	LIEN	DESCRIPTION
C# 7.0 et versions ultérieures		Actuellement non disponible
C# 6.0	Lien	Spécification du langage C# version 6 - Ébauche non officielle : .NET Foundation
C# 5.0	Télécharger le PDF	Norme ECMA-334 5e édition
C# 3.0	Télécharger DOC	Spécification du langage C# version 3.0 : Microsoft Corporation

VERSION	LIEN	DESCRIPTION
C# 2.0	Télécharger le PDF	Norme ECMA-334 4e édition
C# 1.2	Télécharger DOC	Spécification du langage C# version 1.2 : Microsoft Corporation
C# 1.0	Télécharger DOC	Spécification du langage C# version 1.0 : Microsoft Corporation

Version minimale du kit de développement logiciel nécessaire pour prendre en charge toutes les fonctionnalités de langage

Le tableau suivant répertorie les versions minimales du kit de développement logiciel (SDK) avec le compilateur C# qui prend en charge la version de langue correspondante :

VERSION C#	VERSION MINIMALE DU KIT DE DÉVELOPPEMENT LOGICIEL
C# 8.0	Microsoft Visual Studio/Build Tools 2019, version 16,3 ou .NET Core 3,0 SDK
C# 7.3	Microsoft Visual Studio/Build Tools 2017, version 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, version 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, version 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 ou compilateur .NET Framework 4.5 groupé
C# 4	Microsoft Visual Studio/Build Tools 2010 ou compilateur .NET Framework 4.0 groupé
C# 3	Microsoft Visual Studio/Build Tools 2008 ou compilateur .NET Framework 3.5 groupé
C# 2	Microsoft Visual Studio/Build Tools 2005 ou compilateur .NET Framework 2.0 groupé
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 ou regroupé .NET Framework 1,0 du compilateur

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-lib (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-lib** spécifie l'emplacement des assemblys référencés au moyen de l'option -Reference (options du compilateur C#) .

Syntaxe

```
-lib:dir1[,dir2]
```

Arguments

`dir1`

Un répertoire dans lequel le compilateur doit effectuer la recherche si un assembly référencé ne se trouve pas dans le répertoire de travail actuel (le répertoire à partir duquel vous appelez le compilateur) ou dans le répertoire système du common language runtime.

`dir2`

Un ou plusieurs répertoires supplémentaires où rechercher des références d'assembly. Séparez les noms des répertoires supplémentaires par une virgule, sans espace.

Notes

Le compilateur recherche les références d'assembly qui ne sont pas complètes dans l'ordre suivant :

1. Répertoire de travail actuel. Il s'agit du répertoire à partir duquel le compilateur est appelé.
2. Répertoire système du common language runtime.
3. Répertoires spécifiés par **-lib**.
4. Répertoires spécifiés par la variable d'environnement LIB.

Utilisez **-reference** pour spécifier une référence d'assembly.

-lib est additif. Si vous le spécifiez plusieurs fois, il effectue un ajout aux valeurs précédentes.

Au lieu d'utiliser **-lib**, vous pouvez copier dans le répertoire de travail les assemblys nécessaires. De cette façon, vous passez simplement le nom de l'assembly à **-reference**. Vous pouvez ensuite supprimer les assemblys du répertoire de travail. Comme le chemin de l'assembly dépendant n'est pas spécifié dans le manifeste de l'assembly, l'application peut être démarrée sur l'ordinateur cible, et recherche et utilise l'assembly dans le Global Assembly Cache.

Le fait que le compilateur puisse référencer l'assembly n'implique pas que le common language runtime soit en mesure de rechercher et charger l'assembly au moment de l'exécution. Consultez Méthode de localisation des assemblys par le runtime pour plus d'informations sur la façon dont le runtime recherche des assemblys référencés.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la boîte de dialogue **Pages de propriété** du projet.
2. Cliquez sur la page de propriétés **Chemin aux références**.

3. Modifiez le contenu de la zone de liste.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez ReferencePath.

Exemple

Compilez t2.cs pour créer un fichier .exe. Le compilateur recherche les références d'assembly dans le répertoire de travail et dans le répertoire racine du lecteur C.

```
csc -lib:c:\ -reference:t2.dll t2.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-link (Options du compilateur C#)

02/11/2020 • 7 minutes to read • Edit Online

Fait que le compilateur rend disponible pour le projet en cours de compilation les informations de type COM des assemblys spécifiés.

Syntaxe

```
-link:fileList
// -or-
-l:fileList
```

Arguments

`fileList`
Obligatoire. Liste délimitée par des virgules des noms de fichiers d'assembly. Si le nom de fichier contient un espace, placez-le entre des guillemets.

Notes

L'option `-link` vous permet de déployer une application qui a des informations de type incorporées. L'application peut ensuite utiliser des types dans un assembly de runtime qui implémentent les informations de type incorporées sans nécessiter une référence à l'assembly de runtime. Si différentes versions de l'assembly de runtime sont publiées, l'application qui contient les informations de type incorporées peut fonctionner avec les différentes versions sans devoir être recompilée. Pour un exemple, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

L'utilisation de l'option `-link` est particulièrement utile quand vous travaillez avec COM Interop. Vous pouvez incorporer des types COM afin que votre application ne nécessite plus un assembly PIA (Primary Interop Assembly) sur l'ordinateur cible. L'option `-link` indique au compilateur d'incorporer les informations de type COM provenant de l'assembly Interop référencé dans le code compilé résultant. Le type COM est identifié par la valeur du CLSID (GUID). Par conséquent, votre application peut s'exécuter sur un ordinateur cible où les mêmes types COM ont été installés avec les mêmes valeurs de CLSID. Les applications qui automatisent Microsoft Office sont un bon exemple. Étant donné que les applications comme Office gardent habituellement la même valeur de CLSID à travers différentes versions, votre application peut utiliser les types COM référencés dès lors que le .NET Framework 4 ou ultérieur est installé sur l'ordinateur cible, et que votre application utilise les méthodes, propriétés ou événements qui sont inclus dans les types COM référencés.

L'option `-link` incorpore seulement les interfaces, les structures et les délégués. L'incorporation de classes COM n'est pas prise en charge.

> **NOTE**
> Quand vous créez une instance d'un type COM incorporé dans votre code, vous devez créer l'instance à l'aide de l'interface appropriée. Une tentative de création d'une instance d'un type COM incorporé à l'aide de la coclasse provoque une erreur.

Pour définir l'option `-link` dans Visual Studio, ajoutez une référence d'assembly et définissez la propriété `Embed Interop Types` sur **true**. La valeur par défaut pour la propriété `Embed Interop Types` est **false**.

Si vous liez à un assembly COM (Assembly A) qui référence lui-même un autre assembly COM (Assembly B), vous devez également lier à l'Assembly B si une des conditions suivantes est vraie :

- Un type de l'Assembly A hérite d'un type ou implémente une interface de l'Assembly B.
- Un champ, une propriété, un événement ou une méthode qui a un type de retour ou un type de paramètre de l'Assembly B est appelé.

À l'instar de l'option -Reference du compilateur, l' `-link` option du compilateur utilise le fichier réponse CSC. rsp, qui référence les assemblys .net fréquemment utilisés. Utilisez l'option -noconfig du compilateur si vous ne voulez pas que le compilateur utilise le fichier Csc.rsp.

La forme abrégée de `-link` est `-l`.

Types génériques et imbriqués

Les sections suivantes décrivent les limitations de l'utilisation de types génériques dans les applications qui incorporent des types interop.

Interfaces génériques

Vous ne pouvez pas utiliser des interfaces génériques incorporées depuis un assembly d'interopérabilité. Cela est illustré par l'exemple suivant.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

Types ayant des paramètres génériques

Vous ne pouvez pas utiliser des types qui ont un paramètre générique dont le type est incorporé depuis un assembly d'interopérabilité si ce type provient d'un assembly externe. Cette restriction ne s'applique pas aux interfaces. Prenons l'exemple de l'interface Range qui est définie dans l'assembly Microsoft.Office.Interop.Excel. Si une bibliothèque incorpore les types interop de l'assembly Microsoft.Office.Interop.Excel et expose une méthode qui retourne un type générique qui a un paramètre dont le type est l'interface Range, cette méthode doit retourner une interface générique, comme dans l'exemple de code suivant.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

Dans l'exemple suivant, le code client peut appeler la méthode qui retourne l'interface générique IList sans erreur.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

Exemple

Le code suivant compile le fichier source `OfficeApp.cs` et des assemblys de référence à partir de `COMData1.dll` et de `COMData2.dll` pour produire `OfficeApp.exe`.

```
csc -link:COMData1.dll,COMData2.dll -out:OfficeApp.exe OfficeApp.cs
```

Voir aussi

- Options du compilateur C#
- Procédure pas à pas : incorporation de types provenant d'assemblys managés
- -reference (Options du compilateur C#)
- -noconfig (Options du compilateur C#)
- Génération à partir de la ligne de commande avec csc.exe
- Vue d'ensemble de l'interopérabilité

-linkresource (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

Crée un lien vers une ressource .NET dans le fichier de sortie. Le fichier de ressources n'est pas ajouté au fichier de sortie. Cette option diffère de l'option -resource qui incorpore un fichier de ressources dans le fichier de sortie.

Syntaxe

```
-linkresource:filename[,identifier[,accessibility-modifier]]
```

Arguments

`filename`
Fichier de ressources .NET auquel vous souhaitez lier l'assembly.

`identifier` (facultatif)
Nom logique de la ressource. Il s'agit du nom utilisé pour charger la ressource. La valeur par défaut est le nom du fichier.

`accessibility-modifier` (facultatif)
Accessibilité de la ressource : publique ou privée. La valeur par défaut est public.

Notes

Par défaut, les ressources liées sont publiques dans l'assembly quand elles sont créées avec le compilateur C#. Pour rendre les ressources privées, spécifiez le modificateur d'accessibilité `private`. Aucun autre modificateur que `public` ou `private` n'est autorisé.

-linkresource nécessite une option -target autre que **-target:module**.

Si `filename` est un fichier de ressources .net créé, par exemple par Resgen.exe ou dans l'environnement de développement, il est accessible à l'aide des membres de l' System.Resources espace de noms. Pour plus d'informations, consultez System.Resources.ResourceManager. Pour toutes les autres ressources, utilisez les méthodes `GetManifestResource` dans la classe Assembly pour accéder à la ressource au moment de l'exécution.

Le fichier spécifié dans `filename` peut avoir n'importe quel format. C'est le cas, par exemple, si vous voulez qu'une DLL native fasse partie de l'assembly pour qu'elle puisse être installée dans le Global Assembly Cache et accessible à partir du code managé dans l'assembly. Le deuxième des exemples suivants montre comment effectuer cette opération. Vous pouvez effectuer la même opération dans Assembly Linker. Le troisième des exemples suivants montre comment effectuer cette opération. Pour plus d'informations, consultez Al.exe (Assembly Linker) et Utilisation d'assemblys et du Global Assembly Cache.

-linkres est la forme abrégée de **-linkresource**.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Exemple

Compiler `in.cs` et créer un lien vers le fichier de ressources `rf.resource` :

```
csc -linkresource:rf.resource in.cs
```

Exemple

Compiler `A.cs` dans une DLL, créer un lien vers une DLL native N.dll et placer la sortie dans le Global Assembly Cache (GAC). Dans cet exemple, A.dll et N.dll résident dans le GAC.

```
csc -linkresource:N.dll -t:library A.cs
gacutil -i A.dll
```

Exemple

Cet exemple obtient le même résultat, mais en utilisant des options Assembly Linker.

```
csc -t:module A.cs
al -out:A.dll A.netmodule -link:N.dll
gacutil -i A.dll
```

Voir aussi

- Options du compilateur C#
- Al.exe (Assembly Linker)
- Utilisation d'assemblys et du Global Assembly Cache
- Gestion des propriétés des projets et des solutions

-main (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette option spécifie la classe qui contient le point d'entrée du programme dans le cas où plusieurs classes contiennent une méthode **Main**.

Syntaxe

```
-main:class
```

Arguments

`class`
Type contenant la méthode **Main**.
Le nom de classe fourni doit être complet, c'est-à-dire qu'il doit inclure l'espace de noms complet contenant la classe, suivi du nom de classe. Par exemple, si la méthode `Main` se trouve dans la classe `Program` dans l'espace de noms `MyApplication.Core`, l'option du compilateur doit être `-main:MyApplication.Core.Program`.

Notes

Si votre compilation comprend plusieurs types avec une méthode Main, vous pouvez spécifier le type qui contient la méthode **Main** à utiliser comme point d'entrée dans le programme.

Cette option est destinée à être utilisée lors de la compilation d'un fichier *.exe*.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Objet de démarrage**.

   Pour définir cette option du compilateur par programmation, consultez StartupObject.

Pour définir cette option du compilateur en modifiant manuellement le fichier *.csproj*

Vous pouvez définir cette option en modifiant le fichier. csproj et en ajoutant un élément `StartupObject` à l'intérieur de la `PropertyGroup` section. Par exemple :

```
<PropertyGroup>
  ...
  <StartupObject>MyApplication.Core.Program</StartupObject>
</PropertyGroup>
```

Exemple

Compilez `t2.cs` et `t3.cs`, en spécifiant que la **Main** se trouve dans `Test2` :

```
csc t2.cs t3.cs -main:Test2
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-moduleassemblyname (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Spécifie un assembly dont les types non publics sont accessibles par un .netmodule.

Syntaxe

```
-moduleassemblyname:assembly_name
```

Arguments

`assembly_name`
Nom de l'assembly dont les types non publics sont accessibles au fichier .netmodule.

Notes

-moduleassemblyname doit être utilisée au moment de créer un fichier .netmodule et quand les conditions suivantes sont réunies :

- Le fichier .netmodule doit accéder à des types non publics dans un assembly existant.
- Vous connaissez le nom de l'assembly dans lequel le fichier .netmodule sera généré.
- L'assembly existant a accordé un accès d'assembly friend à l'assembly dans lequel le fichier .netmodule sera généré.

Pour plus d'informations sur la création d'un fichier .netmodule, consultez -target:module (Options du compilateur C#).

Pour plus d'informations sur les assemblys friend, consultez Assemblys friend.

Cette option n'est pas disponible dans l'environnement de développement ; elle l'est uniquement au moment de compiler à partir de la ligne de commande.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Exemple

Cet exemple génère un assembly avec un type privé, ce qui octroie un accès d'assembly friend à un assembly appelé csman_an_assembly.

```
// moduleassemblyname_1.cs
// compile with: -target:library
using System;
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo ("csman_an_assembly")]

class An_Internal_Class
{
    public void Test()
    {
        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

Exemple

Cet exemple génère un fichier .netmodule qui accède à un type non public dans l'assembly moduleassemblyname_1.dll. En sachant que ce. netmodule sera intégré à un assembly appelé csman_an_assembly, nous pouvons spécifier **-moduleassemblyname**, ce qui permet au. netmodule d'accéder aux types non publics dans un assembly qui a accordé un accès d'assembly friend à csman_an_assembly.

```
// moduleassemblyname_2.cs
// compile with: -moduleassemblyname:csman_an_assembly -target:module -reference:moduleassemblyname_1.dll
class B {
    public void Test() {
        An_Internal_Class x = new An_Internal_Class();
        x.Test();
    }
}
```

Exemple

Cet exemple de code génère l'assembly csman_an_assembly en faisant référence à l'assembly et au fichier .netmodule générés précédemment.

```
// csman_an_assembly.cs
// compile with: -addmodule:moduleassemblyname_2.netmodule -reference:moduleassemblyname_1.dll
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

An_Internal_Class.Test called

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-noconfig (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-noconfig** indique au compilateur de ne pas compiler avec le fichier csc. rsp, qui se trouve dans le même répertoire que le fichier csc.exe et est chargé à partir de celui-ci.

Syntaxe

```
-noconfig
```

Notes

Le fichier csc. rsp fait référence à tous les assemblys fournis avec .NET Framework. Les références réelles qui sont incluses dans l'environnement de développement Visual Studio .NET varient en fonction du type de projet.

Vous pouvez modifier le fichier csc. RSP et spécifier des options de compilateur supplémentaires qui doivent être incluses dans chaque compilation à partir de la ligne de commande avec csc.exe (à l'exception de l'option **-noconfig**).

Le compilateur traite les options passées à la commande **csc** en dernier. De ce fait, toute option sur la ligne de commande se substitue au paramètre de la même option dans le fichier csc.rsp.

Si vous ne souhaitez pas que le compilateur recherche et utilise les paramètres dans le fichier csc. rsp, spécifiez **-noconfig**.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-nologo (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-nologo** supprime l'affichage de la bannière d'authentification lors du démarrage du compilateur et de l'affichage des messages d'information pendant la compilation.

Syntaxe

```
-nologo
```

Notes

Cette option n'est pas disponible dans l'environnement de développement ; elle l'est uniquement au moment de compiler à partir de la ligne de commande.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-nostdlib (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

-nostdlib empêche l'importation du fichier mscorlib.dll, qui définit l'espace de noms System tout entier.

Syntaxe

```
-nostdlib[+ | -]
```

Notes

Utilisez cette option si vous souhaitez définir ou créer vos propres objets et espace de noms System.

Si vous ne spécifiez pas **-nostdlib**, mscorlib.dll est importé dans votre programme (ce qui équivaut à spécifier **-nostdlib-**). Les options **-nostdlib** et **-nostdlib+** sont équivalentes.

Pour définir cette option de compilateur dans Visual Studio

> **NOTE**
>
> Les instructions suivantes s'appliquent uniquement à Visual Studio 2015 (et versions antérieures). La propriété **ne pas référencer mscorlib.dll** Build n'existe pas dans les versions plus récentes de Visual Studio.

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer** .
3. Cliquez sur le bouton **Avancé** .
4. Modifiez la propriété **Ne pas référencer mscorlib.dll** .

Pour définir cette option du compilateur par programmation

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez NoStdLib.

Voir aussi

- Options du compilateur C#

-nowarn (options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-nowarn** vous permet de désactiver l'affichage d'un ou plusieurs avertissements par le compilateur. Séparez les numéros des avertissements par une virgule.

Syntaxe

```
-nowarn:number1[,number2,...]
```

Arguments

`number1`, `number2`
Numéro de chaque avertissement que le compilateur ne doit pas afficher.

Notes

Vous devez spécifier uniquement la partie numérique de l'identificateur d'avertissement. Par exemple, si vous souhaitez supprimer l'avertissement CS0028, spécifiez `-nowarn:28`.

Le compilateur ignore automatiquement les numéros des avertissements passés à `-nowarn` qui étaient valides dans les versions précédentes, mais qui ont été supprimés dans le compilateur. Par exemple, l'avertissement CS0679 était valide dans le compilateur dans Visual Studio .NET 2002, mais il a été supprimé depuis cette version.

Les avertissements suivants ne peuvent pas être supprimés avec l'option `-nowarn` :

- Avertissement du compilateur (niveau 1) CS2002
- Avertissement du compilateur (niveau 1) CS2023
- Avertissement du compilateur (niveau 1) CS2029

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet. Pour plus de détails, consultez Générer, page du Concepteur de projet (C#).
2. Cliquez sur la page de propriétés **Générer**.
3. Modifiez la propriété **Supprimer les avertissements**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez DelaySign.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions
- Erreurs du compilateur C#

-nowin32manifest (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Utilisez l'option **-nowin32manifest** pour indiquer au compilateur de ne pas incorporer le manifeste de l'application dans le fichier exécutable.

Syntaxe

```
-nowin32manifest
```

Notes

Quand cette option est utilisée, l'application est soumise à la virtualisation sur Windows Vista, sauf si vous fournissez un manifeste de l'application dans un fichier de ressources Win32 ou au cours d'une étape de génération ultérieure.

Dans Visual Studio, définissez cette option dans la page de propriétés **Application** en sélectionnant l'option **Créer une application sans fichier manifeste** dans la zone de liste déroulante **Manifeste**. Pour plus d'informations, consultez page application, concepteur de projets (C#).

Pour plus d'informations sur la création d'un manifeste, consultez -win32manifest (options du compilateur C#).

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-Nullable (options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-Nullable** vous permet de spécifier le contexte Nullable.

Syntaxe

```
-nullable[+ | -]
-nullable:{enable | disable | warnings | annotations}
```

Arguments

`+` | `-`
Si `+` vous spécifiez, ou **-Nullable**, le compilateur active le contexte Nullable. `-` Si vous spécifiez, qui est en vigueur si vous ne spécifiez pas **-Nullable**, désactive le contexte Nullable.

`enable` | `disable` | `warnings` | `annotations`
Spécifie l'option de contexte Nullable. Semblable à `+` ou `-` , pour activer et désactiver, mais permet une plus grande granularité de la spécificité du contexte Nullable. L' `enable` argument, qui est en effet le même que si vous spécifiez **-Nullable**, active le contexte Nullable. La spécification de `disable` désactive le contexte Nullable. Lorsque vous fournissez l' `warnings` argument, **-Nullable : Warnings**, le contexte d'avertissement Nullable est activé. Lors de la spécification de l' `annotations` argument, **-Nullable : annotations**, le contexte d'annotation Nullable est activé.

Notes

L'analyse de Flow permet de déduire la possibilité de valeur null des variables au sein du code exécutable. La possibilité de valeur null déduite d'une variable est indépendante de la possibilité de valeur null déclarée de la variable. Les appels de méthode sont analysés même lorsqu'ils sont omis de manière conditionnelle. Par exemple, Debug.Assert en mode release.

L'appel de méthodes annotées avec les attributs suivants affecte également l'analyse du workflow :

- Conditions préalables simples : AllowNullAttribute et DisallowNullAttribute
- Conditions postérieures simples : MaybeNullAttribute et NotNullAttribute
- Conditions de publication conditionnelles : MaybeNullWhenAttribute et NotNullWhenAttribute
- DoesNotReturnIfAttribute (par exemple, `DoesNotReturnIf(false)` pour Debug.Assert) et DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- Conditions de publication des membres : MemberNotNullAttribute(String) et MemberNotNullAttribute(String[])

IMPORTANT

Le contexte Nullable global ne s'applique pas aux fichiers de code générés. Indépendamment de ce paramètre, le contexte Nullable est *désactivé* pour tout fichier source marqué comme généré. Un fichier est marqué comme généré de quatre façons :

1. Dans le fichier. editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
2. Placez `<auto-generated>` ou `<auto-generated/>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.
3. Démarrer le nom de fichier avec *TemporaryGeneratedFile_*
4. Terminez le nom de fichier avec *. Designer.cs*, *. generated.cs*, *. g.cs* ou *. g.i.cs*.

Les générateurs peuvent s'abonner à l'aide de la `#nullable` directive de préprocesseur.

Pour définir cette option du compilateur dans un projet

Modifiez le fichier *. csproj* pour ajouter la `<Nullable>` balise dans une `Project/PropertyGroup` hiérarchie :

```
<Project Sdk="...">

  <PropertyGroup>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Voir aussi

- Options du compilateur C#
- `#nullable` directive de préprocesseur
- Types références Nullables

-optimize (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-optimize** active ou désactive les optimisations effectuées par le compilateur pour réduire la taille de votre fichier de sortie et le rendre plus rapide et plus efficace.

Syntaxe

```
-optimize[+ | -]
```

Notes

-optimize indique aussi au Common Language Runtime d'optimiser le code au moment de l'exécution.

Par défaut, les optimisations sont désactivées. Pour activer les optimisations, spécifiez **-optimize+**.

Quand vous créez un module destiné à être utilisé par un assembly, utilisez les mêmes paramètres **-optimize** que ceux de l'assembly.

-o est la forme abrégée de **-optimize**.

Il est possible de combiner les options **-optimize** et -debug.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Modifiez la propriété **Optimiser le code**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez Optimize.

Exemple

Compilez `t2.cs` et activez les optimisations du compilateur :

```
csc t2.cs -optimize
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-out (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-out** spécifie le nom du fichier de sortie.

Syntaxe

```
-out:filename
```

Arguments

`filename`
Nom du fichier de sortie créé par le compilateur.

Notes

Sur la ligne de commande, il est possible de spécifier plusieurs fichiers de sortie pour une même compilation. Le compilateur s'attend à trouver un ou plusieurs fichiers de code source à la suite de l'option **-out**. Ensuite, tous les fichiers de code source sont compilés dans le fichier de sortie spécifié par cette option **-out**.

Spécifiez le nom complet et l'extension du fichier que vous voulez créer.

Si vous ne spécifiez pas le nom du fichier de sortie :

- Un fichier .exe prend le nom du fichier de code source qui contient la méthode **Main**.
- Un fichier .dll ou .netmodule prend le nom du premier fichier de code source.

Un fichier de code source ayant servi à compiler un fichier de sortie ne peut pas être utilisé dans une même compilation pour compiler un autre fichier de sortie.

Au moment de générer plusieurs fichiers de sortie lors d'une compilation en ligne de commande, gardez à l'esprit que seul un fichier de sortie peut faire office d'assembly et que seul le premier fichier de sortie spécifié (implicitement ou explicitement avec **-out**) peut être l'assembly.

Les modules générés dans le cadre d'une compilation deviennent des fichiers associés à un assembly également généré pendant la compilation. Utilisez ildasm.exe pour afficher le manifeste d'assembly et identifier les fichiers associés.

L'option de compilateur -out est nécessaire pour faire d'un fichier exe la cible d'un assembly friend. Pour plus d'informations, consultez Assemblys friend.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Nom de l'assembly**.

   Pour définir cette option de compilateur par programmation : OutputFileName est une propriété en lecture seule, qui est déterminée par une combinaison du type de projet (exe, bibliothèque, etc.) et le nom de l'assembly. Il est nécessaire de modifier l'une de ces propriétés (ou les deux) pour définir le nom du fichier de sortie.

Exemple

Compilez `t.cs` et créez le fichier de sortie `t.exe`, puis générez `t2.cs` et créez le fichier de sortie de module `mymodule.netmodule` :

```
csc t.cs -out:mymodule.netmodule -target:module t2.cs
```

Voir aussi

- Options du compilateur C#
- Assemblys friend
- Gestion des propriétés des projets et des solutions

-pathmap (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option de compilateur **-pathmap** spécifie comment mapper des chemins physiques aux noms de chemin source générés en sortie par le compilateur.

Syntaxe

```
-pathmap:path1=sourcePath1,path2=sourcePath2
```

Arguments

`path1` : chemin complet aux fichiers sources dans l'environnement actuel

`sourcePath1` : chemin source remplaçant `path1` dans les fichiers de sortie.

Pour spécifier plusieurs chemins sources mappés, séparez-les par des virgules.

Notes

Le compilateur écrit le chemin source dans sa sortie dans les cas suivants :

1. Le chemin source est remplacé par un argument quand CallerFilePathAttribute est appliqué à un paramètre facultatif.
2. Le chemin source est incorporé dans un fichier PDB.
3. Le chemin du fichier PDB est incorporé dans un fichier exécutable portable (PE).

Cette option mappe chaque chemin physique de l'ordinateur sur lequel le compilateur s'exécute à un chemin correspondant qui doit être écrit dans les fichiers de sortie.

Exemple

Compilez `t.cs` dans le répertoire **C:\work\tests** et mappez ce répertoire à **\publish** dans la sortie :

```
csc -pathmap:C:\work\tests=\publish t.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-pdb (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option de compilateur **-pdb** spécifie le nom et l'emplacement du fichier de symboles de débogage.

Syntaxe

```
-pdb:filename
```

Arguments

`filename`
Nom et emplacement du fichier de symboles de débogage.

Notes

Quand vous spécifiez -debug (Options du compilateur C#), le compilateur crée un fichier .pdb dans le même répertoire que celui dans lequel le compilateur va créer le fichier de sortie (.exe ou .dll) avec un nom de fichier identique à celui du fichier de sortie.

-pdb vous permet de spécifier un nom de fichier autre que celui par défaut et un emplacement pour le fichier .pdb.

Cette option du compilateur ne peut pas être définie dans l'environnement de développement Visual Studio, ni être modifiée par programmation.

Exemple

Compilez `t.cs` et créez un fichier .pdb sous le nom tt.pdb :

```
csc -debug -pdb:tt t.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-platform (Options du compilateur C#)

16/12/2020 • 4 minutes to read • Edit Online

Spécifie la version du CLR (Common Language Runtime) qui peut exécuter l'assembly.

Syntaxe

```
-platform:string
```

Paramètres

`string`
anycpu (valeur par défaut), anycpu32bitpreferred, ARM, x64, x86 ou Itanium.

Notes

- **anycpu** (valeur par défaut) compile votre assembly pour qu'il s'exécute sur n'importe quelle plateforme. Votre application s'exécute en tant que processus 64 bits dans la mesure du possible et repasse en 32 bits quand seul ce mode est disponible.
- **anycpu32bitpreferred** compile votre assembly pour qu'il s'exécute sur n'importe quelle plateforme. Votre application s'exécute en mode 32 bits sur les systèmes qui prennent en charge les applications 64 bits et 32 bits. Vous pouvez spécifier cette option uniquement pour les projets qui ciblent .NET Framework 4,5 ou une version ultérieure.
- **ARM** compile votre assembly pour qu'il s'exécute sur un ordinateur doté d'un processeur ARM (Advanced RISC Machine).
- **ARM64** compile votre assembly pour s'exécuter par le CLR 64 bits sur un ordinateur qui dispose d'un processeur Advanced RISC Machine (ARM) qui prend en charge le jeu d'instructions A64.
- **x64** compile votre assembly pour qu'il soit exécuté par le CLR 64 bits sur un ordinateur qui prend en charge le jeu d'instructions AMD64 ou EM64T.
- **x86** compile votre assembly pour qu'il soit exécuté par le CLR 32 bits x86.
- **Itanium** compile votre assembly pour qu'il soit exécuté par le CLR 64 bits sur un ordinateur doté d'un processeur Itanium.

Sur un système d'exploitation Windows 64 bits :

- Les assemblys compilés avec **-platform:x86** s'exécutent sur le CLR 32 bits fonctionnant sous WOW64.
- Une DLL compilée avec **-platform:anycpu** s'exécute sur le même CLR que le processus dans lequel elle est chargée.
- Les fichiers exécutables compilés avec **-platform:anycpu** s'exécutent sur le CLR 64 bits.
- Les fichiers exécutables compilés avec **-platform:anycpu32bitpreferred** s'exécutent sur le CLR 32 bits.

Le paramètre **anycpu32bitpreferred** est valide uniquement pour l'exécutable (. EXE) et nécessite .NET Framework 4,5 ou une version ultérieure.

Pour plus d'informations sur le développement d'une application s'exécutant sur un système d'exploitation Windows 64 bits, consultez Applications 64 bits.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Modifiez la propriété **plateforme cible** et, pour les projets qui ciblent .NET Framework 4,5 ou une version ultérieure, activez ou désactivez la case à cocher **préférer 32 bits** .

> **NOTE**
> `-platform` n'est pas disponible dans l'environnement de développement de Visual C# Express.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez PlatformTarget.

Exemple

L'exemple suivant montre comment utiliser l'option **-platform** pour préciser que l'application doit être exécutée par le CLR 64 bits sur un système d'exploitation Windows 64 bits.

```
csc -platform:anycpu filename.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-preferreduilang (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

L'option de compilateur `-preferreduilang` vous permet de spécifier la langue dans laquelle le compilateur C# affiche la sortie, comme les messages d'erreur.

Syntaxe

```
-preferreduilang: language
```

Arguments

`language`
Le nom de la langue à utiliser pour la sortie du compilateur.

Notes

Vous pouvez utiliser l'option de compilateur `-preferreduilang` pour spécifier la langue que le compilateur C# doit utiliser pour les messages d'erreur et d'autres types de sortie de ligne de commande. Si le module linguistique de la langue n'est pas installé, le paramètre de langue du système d'exploitation est utilisé et aucune erreur n'est signalée.

```
csc.exe -preferreduilang:ja-JP
```

Spécifications

Voir aussi

- Options du compilateur C#

-publicsign (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

Cette option force le compilateur à appliquer une clé publique sans signer l'assembly. L'option **-publicsign** définit également un bit dans l'assembly qui indique au runtime que le fichier est signé.

Syntaxe

```
-publicsign
```

Arguments

Aucun.

Notes

L'option **-publicsign** nécessite l'utilisation de l'option -keyfile ou -keycontainer. Les options **keyfile** ou **keycontainer** spécifient la clé publique.

Les options **-publicsign** et **-delaysign** s'excluent mutuellement.

Parfois appelée « fausse signature » ou « signature OSS », la signature publique inclut la clé publique dans un assembly de sortie et définit l'indicateur « signé ». Toutefois, elle ne signe pas l'assembly avec une clé privée. Cette approche est utile dans le cadre de projets open source. Vous pouvez en effet générer des assemblys compatibles avec les assemblys publiés « totalement signés », même si vous n'avez pas accès à la clé privée utilisée pour les signer. Puisque pratiquement aucun consommateur ne doit vérifier si l'assembly est totalement signé, ces assemblys générés publiquement peuvent être utilisés dans la quasi-totalité des scénarios employant des assemblys totalement signés.

Pour définir cette option du compilateur dans un fichier csproj

Ouvrez le fichier. csproj pour un projet, puis ajoutez l'élément suivant :

```
<PublicSign>true</PublicSign>
```

Voir aussi

- Option -delaysign du compilateur C#
- Option -keyfile du compilateur C#
- Option -keycontainer du compilateur C#
- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-recurse (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option -recurse permet de compiler des fichiers de code source dans tous les répertoires enfants du répertoire spécifié (dir) ou du répertoire du projet.

Syntaxe

```
-recurse:[dir\]file
```

Arguments

`dir` (facultatif)
Répertoire dans lequel vous voulez commencer la recherche. Si ce répertoire n'est pas spécifié, la recherche commence dans le répertoire du projet.

`file`
Le ou les fichiers à rechercher. Les caractères génériques sont autorisés.

Notes

L'option **-recurse** vous permet de compiler des fichiers de code source dans tous les répertoires enfants du répertoire spécifié (`dir`) ou du répertoire du projet.

Vous pouvez utiliser des caractères génériques dans un nom de fichier pour compiler tous les fichiers correspondants dans le répertoire du projet sans utiliser **-recurse**.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Exemple

Compile tous les fichiers C# qui se trouvent dans le répertoire actif :

```
csc *.cs
```

Compile tous les fichiers C# du répertoire dir1\dir2 et de tous les sous-répertoires qui se situent en dessous de celui-ci, puis génère dir2.dll :

```
csc -target:library -out:dir2.dll -recurse:dir1\dir2\*.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-reference (Options du compilateur C#)

16/12/2020 • 6 minutes to read • Edit Online

L'option **-reference** indique au compilateur d'importer des informations de type public dans le fichier spécifié du projet actuel, ce qui vous permet de référencer les métadonnées des fichiers d'assembly spécifiés.

Syntaxe

```
-reference:[alias=]filename
-reference:filename
```

Arguments

`filename`
Nom d'un fichier comportant un manifeste d'assembly. Pour importer plusieurs fichiers, incluez une option **-reference** distincte pour chaque fichier.

`alias`
Un identificateur C# valide représentant un espace de noms racine qui contient tous les espaces de noms dans l'assembly.

Notes

Pour importer à partir de plusieurs fichiers, incluez une option **-reference** pour chaque fichier.

Les fichiers que vous importez doivent contenir un manifeste ; le fichier de sortie doit être compilé avec une option -target autre que -target:module.

-r est la forme abrégée de **-reference**.

Utilisez -addmodule pour importer les métadonnées d'un fichier de sortie qui ne contient pas de manifeste d'assembly.

Si vous référencez un assembly (Assembly A) qui référence un autre assembly (Assembly B), vous devez référencer l'Assembly B si :

- Un type que vous utilisez de l'Assembly A hérite d'un type ou implémente une interface de l'Assembly B.
- Vous appelez un champ, une propriété, un événement ou une méthode qui a un type de retour ou un type de paramètre de l'Assembly B.

Utilisez -lib pour spécifier le répertoire dans lequel se trouvent une ou plusieurs références d'assembly. La rubrique **-lib** décrit également les répertoires dans lesquels le compilateur recherche les assemblys.

Pour que le compilateur reconnaisse un type dans un assembly, et non dans un module, il doit être forcé à résoudre le type, ce que vous pouvez faire en définissant une instance du type. Il existe d'autres moyens de résoudre les noms de types dans un assembly pour le compilateur : par exemple, si vous héritez d'un type dans un assembly, le nom du type est ensuite reconnu par le compilateur.

Il est parfois nécessaire de référencer deux versions différentes du même composant dans un assembly. Pour ce faire, utilisez la sous-option d'alias du commutateur **-reference** pour chaque fichier afin de faire la différence entre les deux fichiers. Cet alias est utilisé comme qualificateur pour le nom du composant et se traduit par le

composant dans l'un des fichiers.

Le fichier de réponse csc (.rsp), qui référence les assemblys .NET Framework couramment utilisés, est utilisé par défaut. Utilisez -noconfig si vous ne voulez pas que le compilateur utilise csc.rsp.

> **NOTE**
>
> Dans Visual Studio, utilisez la boîte de dialogue **Ajouter une référence**. Pour plus d'informations, consultez Guide pratique pour ajouter ou supprimer des références à l'aide du gestionnaire de références. Pour garantir un comportement équivalent selon que vous ajoutez des références à l'aide de `-reference` ou à l'aide de la boîte de dialogue **Ajouter une référence**, définissez la propriété **Incorporer les types interop** sur **False** pour l'assembly que vous ajoutez. La valeur par défaut de la propriété est **True**.

Exemple

Cet exemple montre comment utiliser la fonctionnalité de l'alias extern.

Vous compilez le fichier source et importez les métadonnées à partir de `grid.dll` et `grid20.dll`, qui ont été compilés au préalable. Les deux DLL contiennent des versions séparées du même composant et vous devez utiliser deux **-reference** avec des options d'alias pour compiler le fichier source. Les options ressemblent à ce qui suit :

```
-reference:GridV1=grid.dll -reference:GridV2=grid20.dll
```

Cela permet de configurer les alias externes `GridV1` et `GridV2`, que vous utilisez dans votre programme à l'aide d'une instruction `extern` :

```
extern alias GridV1;
extern alias GridV2;
// Using statements go here.
```

Une fois cette opération effectuée, vous pouvez référencer le contrôle de grille depuis `grid.dll` en faisant précéder le nom du contrôle par `GridV1`, comme ceci :

```
GridV1::Grid
```

De plus, vous pouvez référencer le contrôle de grille depuis `grid20.dll` en faisant précéder le nom du contrôle par `GridV2`, comme ceci :

```
GridV2::Grid
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-refout (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

L'option **-refout** spécifie un chemin de fichier où l'assembly de référence doit être généré. Cela se traduit par `metadataPeStream` dans l'API Emit. Cette option correspond à la propriété de projet ProduceReferenceAssembly de MSBuild.

Syntaxe

```
-refout:filepath
```

Arguments

`filepath` Chemin de l'assembly de référence. Il doit généralement correspondre à celui de l'assembly principal. La convention recommandée (utilisée par MSBuild) consiste à placer l'assembly de référence dans un sous-dossier « ref/ » par rapport à l'assembly principal.

Remarques

Les assemblys de référence sont un type spécial d'assembly qui ne contient que la quantité minimale de métadonnées requises pour représenter la surface de l'API publique de la bibliothèque. Elles incluent des déclarations pour tous les membres qui sont significatifs lors du référencement d'un assembly dans les outils de génération, mais excluent toutes les implémentations de membres et les déclarations de membres privés qui n'ont aucun impact observable sur leur contrat d'API. Pour plus d'informations, consultez références des assemblys dans le guide .net.

Les `-refout` options et s'excluent `-refonly` mutuellement.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-refonly (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-refonly** indique qu'un assembly de référence doit être généré à la place d'un assembly d'implémentation, en tant que sortie principale. Le paramètre `-refonly` désactive sans assistance la génération de fichiers PDB, car les assemblys de référence ne peuvent pas être exécutés. Cette option correspond à la propriété de projet ProduceOnlyReferenceAssembly de MSBuild.

Syntaxe

```
-refonly
```

Notes

Les assemblys de référence sont un type spécial d'assembly qui ne contient que la quantité minimale de métadonnées requises pour représenter la surface de l'API publique de la bibliothèque. Elles incluent des déclarations pour tous les membres qui sont significatifs lors du référencement d'un assembly dans les outils de génération, mais excluent toutes les implémentations de membres et les déclarations de membres privés qui n'ont aucun impact observable sur leur contrat d'API. Pour plus d'informations, consultez références des assemblys dans le guide .net.

Les `-refonly` options et s'excluent `-refout` mutuellement.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-resource (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Incorpore la ressource spécifiée dans le fichier de sortie.

Syntaxe

```
-resource:filename[,identifier[,accessibility-modifier]]
```

Arguments

`filename`
Fichier de ressources .NET que vous souhaitez incorporer dans le fichier de sortie.

`identifier` (facultatif)
Nom logique de la ressource. Il s'agit du nom utilisé pour charger la ressource. La valeur par défaut est le nom du fichier.

`accessibility-modifier` (facultatif)
Accessibilité de la ressource : publique ou privée. La valeur par défaut est public.

Notes

Utilisez -linkresource pour lier une ressource à un assembly sans ajouter le fichier de ressources au fichier de sortie.

Par défaut, les ressources sont publiques dans l'assembly quand elles sont créées à l'aide du compilateur C#. Pour rendre les ressources privées, spécifiez le modificateur d'accessibilité `private`. Aucune accessibilité autre `public` ou `private` n'est autorisée.

Si `filename` est un fichier de ressources .net créé, par exemple par Resgen.exe ou dans l'environnement de développement, il est accessible à l'aide des membres de l' System.Resources espace de noms. Pour plus d'informations, consultez System.Resources.ResourceManager. Pour toutes les autres ressources, utilisez les méthodes `GetManifestResource` dans la classe Assembly pour accéder à la ressource au moment de l'exécution.

-res est la forme abrégée de **-resource**.

L'ordre des ressources dans le fichier de sortie est déterminé par l'ordre spécifié sur la ligne de commande.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ajoutez un fichier de ressources à votre projet.
2. Sélectionnez le fichier que vous souhaitez incorporer dans l'**Explorateur de solutions**.
3. Sélectionnez **Action de génération** pour le fichier dans la fenêtre **Propriétés**.
4. Définissez **Action de génération** sur **Ressource incorporée**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez BuildAction.

Exemple

Compilez `in.cs` et attachez le fichier de ressources `rf.resource` :

```
csc -resource:rf.resource in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-subsystemversion (Options du compilateur C#)

02/11/2020 • 3 minutes to read • Edit Online

Spécifie la version minimale du sous-système sur lequel le fichier exécutable généré peut s'exécuter, déterminant ainsi les versions de Windows sur lesquelles le fichier exécutable peut s'exécuter. En règle générale, cette option garantit que le fichier exécutable peut tirer parti de fonctionnalités de sécurité particulières qui ne sont pas disponibles avec des versions antérieures de Windows.

> **NOTE**
>
> Pour spécifier le sous-système lui-même, utilisez l'option du compilateur -target.

Syntaxe

```
-subsystemversion:major.minor
```

Paramètres

`major.minor`

Version minimale requise du sous-système, telle qu'elle est exprimée dans une notation par points pour les versions majeure et mineure. Par exemple, vous pouvez spécifier qu'une application ne peut pas s'exécuter sur un système d'exploitation antérieur à Windows 7 si vous définissez la valeur de cette option sur 6.01, comme décrit dans le tableau plus loin dans cette rubrique. Vous devez spécifier les valeurs de `major` et `minor` en tant qu'entiers.

Les zéros à gauche du numéro de version `minor` ne modifient pas la version, contrairement aux zéros de droite. Par exemple, 6.1 et 6.01 font référence à la même version, mais 6.10 fait référence à une version différente. Nous vous recommandons d'exprimer la version mineure avec deux chiffres pour éviter toute confusion.

Remarques

Le tableau suivant répertorie les versions courantes de sous-système de Windows.

VERSION DE WINDOWS	VERSION DE SOUS-SYSTÈME
Windows 2000	5,00
Windows XP	5,01
Windows Server 2003	5.02
Windows Vista	6,00
Windows 7	6.01
Windows Server 2008	6.01

VERSION DE WINDOWS	VERSION DE SOUS-SYSTÈME
Windows 8	6.02

Valeurs par défaut

La valeur par défaut de l'option du compilateur **-subsystemversion** dépend des conditions répertoriées dans la liste suivante :

- La valeur par défaut est 6.02 si l'une quelconque des options du compilateur de la liste suivante est définie :
  - -target:appcontainerexe
  - -target:winmdobj
  - -platform:arm
- La valeur par défaut est 6.00 si vous utilisez MSBuild, si vous ciblez .NET Framework 4.5 et si vous n'avez défini aucune des options du compilateur spécifiées plus haut dans cette liste.
- La valeur par défaut est 4.00 si aucune des conditions précédentes n'est vraie.

Définition de cette option

Pour définir l'option du compilateur **-subsystemversion** dans Visual Studio, vous devez ouvrir le fichier .csproj et spécifier une valeur pour la propriété `SubsystemVersion` dans le code XML MSBuild. Vous ne pouvez pas définir cette option dans l'environnement IDE de Visual Studio. Pour plus d'informations, consultez « Valeurs par défaut » plus haut dans cette rubrique ou Propriétés communes des projets MSBuild.

Voir aussi

- Options du compilateur C#

-target (Options du compilateur C#)

02/11/2020 • 2 minutes to read • Edit Online

L'option du compilateur **-target** peut être spécifiée dans l'une des formes suivantes :

-target:appcontainerexe
Pour créer un fichier. exe pour les applications du Windows 8. x Store.

-target:exe
Pour créer un fichier .exe.

-target:library
Pour créer une bibliothèque de code.

-target:module
Pour créer un module.

-target:winexe
Pour créer un programme Windows.

-target:winmdobj
Pour créer un fichier .winmdobj intermédiaire.

À moins que vous ne spécifiiez **-target : module**, **-target** entraîne le placement d'un manifeste d'assembly .NET Framework dans un fichier de sortie. Pour plus d'informations, consultez assemblys dans .net et attributs communs.

Le manifeste de l'assembly est placé dans le premier fichier de sortie .exe dans la compilation ou dans le premier fichier DLL, en l'absence de fichier de sortie .exe. Par exemple, dans la ligne de commande suivante, le manifeste est placé dans `1.exe` :

```
csc -out:1.exe t1.cs -out:2.netmodule t2.cs
```

Le compilateur crée un seul manifeste d'assembly par compilation. Les informations sur tous les fichiers d'une compilation sont placées dans le manifeste de l'assembly. Tous les fichiers de sortie sauf ceux créés avec **-target : module** peuvent contenir un manifeste d'assembly. Lorsque vous générez plusieurs fichiers de sortie sur la ligne de commande, un seul manifeste d'assembly peut être créé et il doit aller dans le premier fichier de sortie spécifié sur la ligne de commande. Quel que soit le premier fichier de sortie (**-target:exe**, **-target:winexe**, **-target:library** ou **-target:module**), tous les autres fichiers de sortie générés dans la même compilation doivent être des modules (**-target:module**).

Si vous créez un assembly, vous pouvez indiquer que tout ou partie de votre code est conforme CLS avec l'attribut CLSCompliantAttribute.

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

[System.CLSCompliant(false)]   // specify compliance for an element
public class TestClass
{
    public static void Main() {}
}
```

Pour plus d'informations sur la définition de cette option de compilateur par programmation, consultez OutputType.

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions
- -subsystemversion (Options du compilateur C#)

-target:appcontainerexe (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Si vous utilisez l'option du compilateur **-target:appcontainerexe**, le compilateur crée un fichier exécutable Windows (.exe) qui doit être exécuté dans un conteneur d'application. Cette option est équivalente à -target : winexe, mais elle est conçue pour les applications du Windows 8. x Store.

Syntaxe

```
-target:appcontainerexe
```

Notes

Pour exiger que l'application s'exécute dans un conteneur d'application, cette option définit un bit dans le fichier Portable Executable (PE). Lorsque ce bit est défini, une erreur se produit si la méthode CreateProcess tente de lancer l'exécutable en dehors d'un conteneur d'application.

À moins que vous utilisiez l'option -out, le fichier de sortie prend le nom du fichier d'entrée qui contient la méthode Main.

Quand vous spécifiez cette option à une invite de commandes, tous les fichiers jusqu'à l'option **-out** ou **-target** suivante sont utilisés pour créer le fichier exécutable.

Pour définir cette option du compilateur dans l'IDE

1. Dans l'**Explorateur de solutions**, ouvrez le menu contextuel de votre projet et choisissez **Propriétés**.
2. Sous l'onglet **Application**, dans la liste **Type de sortie**, choisissez **Application Windows Store**.

   Cette option est disponible uniquement pour les modèles d'application du Windows 8. x Store.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez OutputType.

Exemple

La commande suivante compile `filename.cs` dans un fichier exécutable Windows qui peut être exécuté uniquement dans un conteneur d'application.

```
csc -target:appcontainerexe filename.cs
```

Voir aussi

- -Target (options du compilateur C#)
- -target : winexe (options du compilateur C#)
- Options du compilateur C#

-target:exe (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-target:exe** indique au compilateur de créer une application console exécutable (EXE).

Syntaxe

```
-target:exe
```

Notes

L'option **-target:exe** est activée par défaut. Le fichier exécutable est créé avec l'extension .exe.

Utilisez -target:winexe pour créer un programme exécutable Windows.

À moins que l'option -out spécifie autre chose, le fichier de sortie prend le nom du fichier d'entrée qui contient la méthode Main.

Quand ils sont spécifiés dans la ligne de commande, tous les fichiers jusqu'à l'option **-out** ou **-target:module** suivante sont utilisés pour créer le fichier .exe.

Une seule et unique méthode **Main** est requise dans les fichiers de code source qui sont compilés dans un fichier .exe. L'option de compilateur -main vous permet de spécifier la classe contenant la méthode **Main**, au cas où votre code possède plusieurs classes avec une méthode **Main**.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Type de sortie**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez OutputType.

Exemple

Chacune des lignes de commande suivantes compile `in.cs` et crée `in.exe` :

```
csc -target:exe in.cs
csc in.cs
```

Voir aussi

- -Target (options du compilateur C#)
- Options du compilateur C#

-target:library (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-target:library** indique au compilateur de créer une bibliothèque de liens dynamiques (DLL) à la place d'un fichier exécutable (EXE).

Syntaxe

```
-target:library
```

Notes

La DLL est créée avec l'extension .dll.

Sauf spécification contraire avec l'option -out, le fichier de sortie prend le nom du premier fichier d'entrée.

Quand ils sont spécifiés dans la ligne de commande, tous les fichiers jusqu'à l'option **-out** ou **-target:module** suivante sont utilisés pour créer le fichier .dll.

Lors de la création d'un fichier .dll, une méthode Main n'est pas obligatoire.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Type de sortie**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez OutputType.

Exemple

Compilez `in.cs`, en créant `in.dll` :

```
csc -target:library in.cs
```

Voir aussi

- -Target (options du compilateur C#)
- Options du compilateur C#

-target:module (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Cette option empêche le compilateur de générer un manifeste d'assembly.

Syntaxe

```
-target:module
```

Notes

Par défaut, le fichier de sortie créé en effectuant une compilation avec cette option porte l'extension .netmodule.

Un fichier qui n'a pas de manifeste d'assembly ne peut pas être chargé par le Runtime .NET. Cependant, un tel fichier peut être incorporé dans le manifeste d'un assembly au moyen de -addmodule.

Si plusieurs modules sont créés dans une seule compilation, les types internal d'un module sont accessibles à d'autres modules dans la compilation. Quand le code d'un module référence des types `internal` dans un autre module, les deux modules doivent être incorporés dans un manifeste d'assembly au moyen de **-addmodule**.

La création d'un module n'est pas prise en charge dans l'environnement de développement Visual Studio.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez OutputType.

Exemple

Compilez `in.cs`, en créant `in.netmodule` :

```
csc -target:module in.cs
```

Voir aussi

- -Target (options du compilateur C#)
- Options du compilateur C#

-target:winexe (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-target : winexe** force le compilateur à créer un programme Windows exécutable (exe).

Syntaxe

```
-target:winexe
```

Notes

Le fichier exécutable est créé avec l'extension .exe. Un programme Windows est un programme qui fournit une interface utilisateur à partir de la bibliothèque .NET ou avec les API Windows.

Utilisez -target:exe pour créer une application console.

À moins que l'option -out spécifie autre chose, le fichier de sortie prend le nom du fichier d'entrée qui contient la méthode Main.

Quand ils sont spécifiés dans la ligne de commande, tous les fichiers jusqu'à l'option **-out** ou -target suivante sont utilisés pour créer le programme Windows.

Une seule et unique méthode **Main** est requise dans les fichiers de code source qui sont compilés dans un fichier .exe. L'option -main vous permet de spécifier la classe contenant la méthode **Main**, au cas où votre code possède plusieurs classes avec une méthode **Main**.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Type de sortie**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez OutputType.

Exemple

Compilez `in.cs` en un programme Windows :

```
csc -target:winexe in.cs
```

Voir aussi

- -Target (options du compilateur C#)
- Options du compilateur C#

-target:winmdobj (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

Si vous utilisez l'option du compilateur **-target:winmdobj**, le compilateur crée un fichier .winmdobj intermédiaire que vous pouvez convertir en fichier binaire (.winmd) Windows Runtime. Le fichier .winmd peut ensuite être consommé par des programmes JavaScript et C++, en plus des programmes en langage managé.

Syntaxe

```
-target:winmdobj
```

Notes

Le paramètre **winmdobj** signale au compilateur qu'un module intermédiaire est requis. En réponse, Visual Studio compile la bibliothèque de classes C# comme fichier .winmdobj. Le fichier .winmdobj peut ensuite être acheminé via l'outil d'exportation WinMDExp pour produire un fichier de métadonnées Windows (.winmd). Le fichier .winmd contient à la fois le code de la bibliothèque d'origine et les métadonnées WinMD utilisées par JavaScript ou C++ et par le Windows Runtime.

La sortie d'un fichier qui est compilé à l'aide de l'option du compilateur **-target:winmdobj** est conçue pour être utilisée uniquement comme entrée pour l'outil d'exportation WimMDExp ; le fichier .winmdobj proprement dit n'est pas référencé directement.

À moins que vous n'utilisiez l'option -out, le fichier de sortie adopte le nom du premier fichier d'entrée. Une méthode Main n'est pas nécessaire.

Si vous spécifiez l'option -target:winmdobj à une invite de commandes, tous les fichiers jusqu'à l'option **-out** ou -target:module suivante sont utilisés pour créer le programme Windows.

Pour définir cette option du compilateur dans l'IDE de Visual Studio pour une application Windows Store

1. Dans l'**Explorateur de solutions**, ouvrez le menu contextuel de votre projet et choisissez **Propriétés**.
2. Choisissez l'onglet **Application**.
3. Dans la liste **Type de sortie**, choisissez **Fichier WinMD**.

   L'option de **fichier WinMD** est disponible uniquement pour les modèles d'application du Windows 8. x Store.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez OutputType.

Exemple

La commande suivante compile `filename.cs` dans un fichier .winmdobj intermédiaire.

```
csc -target:winmdobj filename.cs
```

Voir aussi

- -Target (options du compilateur C#)
- Options du compilateur C#

-unsafe (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option de compilateur **-unsafe** permet la compilation de code utilisant le mot clé unsafe.

Syntaxe

```
-unsafe
```

Notes

Pour plus d'informations sur le code unsafe, consultez Pointeurs et code unsafe.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Cochez la case **Autoriser le code unsafe**.

Pour ajouter cette option dans un fichier csproj

Ouvrez le fichier .csproj d'un projet et ajoutez les éléments suivants :

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez AllowUnsafeBlocks.

Exemple

Compilez `in.cs` selon le mode non sécurisé :

```
csc -unsafe in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-utf8output (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-utf8output** affiche la sortie du compilateur en utilisant un encodage UTF-8.

Syntaxe

```
-utf8output
```

Notes

Dans certaines configurations internationales, la sortie du compilateur ne peut pas s'afficher correctement dans la console. Dans ce cas, utilisez **-utf8output** et redirigez la sortie du compilateur vers un fichier.

Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Voir aussi

- Options du compilateur C#

-warn (Options du compilateur C#)

16/12/2020 • 3 minutes to read • Edit Online

L'option **-warn** spécifie le niveau d'avertissement que le compilateur doit afficher.

Syntaxe

```
-warn:option
```

Arguments

`option`

Niveau d'avertissement que vous souhaitez afficher pour la compilation : les nombres inférieurs affichent uniquement les avertissements ayant un niveau de gravité élevé ; les nombres supérieurs affichent davantage d'avertissements. La valeur doit être zéro ou un entier positif :

NIVEAU D'AVERTISSEMENT	SIGNIFICATION
0	Désactive l'émission de tous les messages d'avertissement.
1	Affiche les messages d'avertissement graves.
2	Affiche les avertissements de niveau 1, ainsi que certains avertissements moins graves, tels que les avertissements concernant le masquage de membres de classe.
3	Affiche les avertissements de niveau 2, ainsi que certains avertissements moins graves, tels que les avertissements concernant les expressions toujours évaluées à `true` ou `false`.
4 (valeur par défaut)	Affiche les avertissements de niveau 3, ainsi que les avertissements d'information.
5	Affiche les avertissements de niveau 4 et des avertissements supplémentaires du compilateur fournis avec C# 9,0.
Supérieur à 5	Toute valeur supérieure à 5 sera traitée comme 5. En général, vous placez une valeur élevée arbitraire (par exemple, `9999`) pour vous assurer que vous avez toujours tous les avertissements si le compilateur est mis à jour avec de nouveaux niveaux d'avertissement.

Notes

Pour obtenir des informations sur une erreur ou un avertissement, vous pouvez rechercher le code d'erreur dans l'index de l'aide. Il existe d'autres façons d'obtenir des informations sur une erreur ou un avertissement, qui sont décrites dans Erreurs du compilateur C#.

Utilisez -warnaserror pour traiter tous les avertissements comme des erreurs. Utilisez -nowarn pour désactiver

certains avertissements.

-w est la forme abrégée de **-warn**.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Modifiez la propriété **Niveau d'avertissement**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez WarningLevel.

Exemple

Compilez `in.cs` et faites en sorte que le compilateur n'affiche que les avertissements de niveau 1 :

```
csc -warn:1 in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-warnaserror (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-warnaserror+** considère tous les avertissements comme des erreurs.

Syntaxe

```
-warnaserror[+ | -][:warning-list]
```

Notes

Les messages qui d'ordinaire auraient été signalés comme des avertissements s'affichent en tant qu'erreurs, et le processus de génération est arrêté (aucun fichier de sortie n'est généré).

Comme **-warnaserror** est activé par défaut, les avertissements n'empêchent pas la génération d'un fichier de sortie. Du fait de **-warnaserror**, qui est identique à **-warnaserror+**, les avertissements sont considérés comme des erreurs.

Le cas échéant, si vous voulez que seuls certains avertissements spécifiques soient considérés comme des erreurs, vous pouvez spécifier une liste séparée par des virgules qui liste les numéros d'avertissement à considérer comme des erreurs. Le jeu de tous les avertissements de possibilité de valeur NULL peut être spécifié avec le raccourci **Nullable** .

Utilisez -warn pour spécifier le niveau des avertissements que le compilateur doit afficher. Utilisez -nowarn pour désactiver certains avertissements.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Générer**.
3. Modifiez la propriété **Considérer les avertissements comme des erreurs**.

Pour définir cette option du compilateur par programmation, consultez TreatWarningsAsErrors.

Exemple

Compilez `in.cs` et faites en sorte que le compilateur n'affiche aucun avertissement :

```
csc -warnaserror in.cs
csc -warnaserror:642,649,652,nullable in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-win32icon (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-win32icon** insère un fichier .ico dans le fichier de sortie, ce qui lui donne l'apparence souhaitée dans l'Explorateur de fichiers.

Syntaxe

```
-win32icon:filename
```

Arguments

`filename`
Fichier .ico que vous voulez ajouter à votre fichier de sortie.

Notes

Un fichier .ico peut être créé avec le compilateur de ressources. Le compilateur de ressources est appelé quand vous compilez un programme Visual C++ ; un fichier .ico est alors créé à partir du fichier .rc.

Consultez -linkresource pour référencer ou -resource pour attacher un fichier de ressources .NET Framework. Consultez -win32res pour importer un fichier .res.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez les pages **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Modifiez la propriété **Icône d'application**.

Pour plus d'informations sur la façon de définir cette option du compilateur par programmation, consultez ApplicationIcon.

Exemple

Compilez `in.cs` et attachez un fichier .ico `rf.ico` afin de générer le fichier `in.exe` :

```
csc -win32icon:rf.ico in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

-win32manifest (Options du compilateur C#)

16/12/2020 • 4 minutes to read • Edit Online

Utilisez l'option **-win32manifest** pour spécifier un fichier manifeste d'application Win32 défini par l'utilisateur à incorporer dans le fichier exécutable portable (PE) d'un projet.

Syntaxe

```
-win32manifest: filename
```

Arguments

`filename`

Nom et emplacement du fichier manifeste personnalisé.

Notes

Par défaut, le compilateur Visual C# incorpore un manifeste d'application qui spécifie le niveau d'exécution requis « asInvoker ». Il crée le manifeste dans le même dossier que celui dans lequel l'exécutable est généré, normalement le dossier bin\Debug ou bin\Release quand vous utilisez Visual Studio. Pour fournir un manifeste personnalisé, par exemple pour spécifier le niveau d'exécution requis « highestAvailable » ou « requireAdministrator », utilisez cette option pour indiquer le nom du fichier.

> **NOTE**
>
> Cette option et l'option -win32res (Options du compilateur C#) s'excluent mutuellement. Si vous tentez d'utiliser ces deux options dans la même ligne de commande, une erreur de build se produira.

Une application sans manifeste d'application pour spécifier le niveau d'exécution requis est soumise à une virtualisation des fichiers/registres sous la fonctionnalité de contrôle de compte d'utilisateur de Windows. Pour plus d'informations, consultez Contrôle de compte d'utilisateur.

Votre application sera soumise à la virtualisation si l'une de ces conditions est vérifiée :

- Vous utilisez l'option **-nowin32manifest** sans fournir de manifeste au cours d'une étape de génération ultérieure ou en tant que partie du fichier de ressources Windows (.res) à l'aide de l'option **-win32res**.
- Vous fournissez un manifeste personnalisé qui ne spécifie pas le niveau d'exécution requis.

Visual Studio crée un fichier .manifest par défaut et le stocke dans les répertoires de débogage et de mise en production avec le fichier exécutable. Vous pouvez ajouter un manifeste personnalisé en en créant un dans un éditeur de texte et en ajoutant le fichier au projet. Vous pouvez également cliquer avec le bouton droit sur l'icône **Projet** de l'**Explorateur de solutions**, cliquer sur **Ajouter un nouvel élément**, puis sur **Fichier manifeste d'application**. Une fois que vous avez ajouté votre fichier manifeste nouveau ou existant, il apparaît dans la liste déroulante **Manifeste**. Pour plus d'informations, consultez page application, concepteur de projets (C#).

Vous pouvez fournir le manifeste d'application en tant qu'étape après génération personnalisée ou en tant que partie d'un fichier de ressources Win32 à l'aide de l'option -nowin32manifest (Options du compilateur C#). Utilisez cette même option pour que votre application soit soumise à la virtualisation des fichiers ou des registres dans Windows Vista. Cela empêche le compilateur de créer et d'incorporer un manifeste par défaut dans le fichier

exécutable portable (PE).

Exemple

L'exemple suivant présente le manifeste par défaut que le compilateur Visual C# insère dans un fichier PE.

> **NOTE**
>
> Le compilateur ajoute un nom d'application standard « MyApplication.app » au fichier xml. Cela permet aux applications de s'exécuter dans Windows Server 2003 Service Pack 3.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Voir aussi

- Options du compilateur C#
- -nowin32manifest (Options du compilateur C#)
- Gestion des propriétés des projets et des solutions

-win32res (Options du compilateur C#)

16/12/2020 • 2 minutes to read • Edit Online

L'option **-win32res** insère une ressource Win32 dans le fichier de sortie.

Syntaxe

```
-win32res:filename
```

Arguments

`filename`
Fichier de ressources que vous voulez ajouter à votre fichier de sortie.

Notes

Un fichier de ressources Win32 peut être créé avec le compilateur de ressources. Le compilateur de ressources est appelé lorsque vous compilez un programme Visual C++ ; un fichier .res est alors créé à partir du fichier .rc.

Une ressource Win32 peut contenir des informations sur la version ou le fichier bitmap (icône) qui permettent d'identifier votre application dans l'Explorateur de fichiers. Si vous ne spécifiez pas l'option **-win32res**, le compilateur génère des informations de version en fonction de la version de l'assembly.

Consultez -linkresource pour référencer ou -resource pour attacher un fichier de ressources .NET Framework.

Pour définir cette option du compilateur dans l'environnement de développement Visual Studio

1. Ouvrez la page **Propriétés** du projet.
2. Cliquez sur la page de propriétés **Application**.
3. Cliquez sur le bouton **Fichier de ressources** et choisissez un fichier à l'aide de la zone de liste modifiable.

Exemple

Compilez `in.cs` et attachez un fichier de ressources Win32 `rf.res` afin de générer le fichier `in.exe` :

```
csc -win32res:rf.res in.cs
```

Voir aussi

- Options du compilateur C#
- Gestion des propriétés des projets et des solutions

Erreurs du compilateur C#

02/11/2020 • 2 minutes to read • Edit Online

Certaines erreurs du compilateur C# possèdent des rubriques correspondantes qui expliquent pourquoi l'erreur est générée et, dans certains cas, comment corriger l'erreur. Suivez l'une des étapes suivantes pour savoir si une aide est fournie pour un message d'erreur donné.

- Trouvez le numéro d'erreur (par exemple, CS0029) dans la fenêtre Sortie, puis recherchez-le sur Microsoft Docs.
- Choisissez le numéro d'erreur (par exemple, CS0029) dans la Fenêtre Sortie, puis appuyez sur la touche F1.
- Entrez le numéro d'erreur dans la zone **Rechercher** de l'index.

Si aucune de ces étapes n'aboutit à des informations sur l'erreur, envoyez des commentaires en bas de cette page en indiquant le numéro ou le texte de l'erreur.

Pour plus d'informations sur la configuration des options d'erreur et d'avertissement en C#, consultez la Page Générer, Concepteur de projets (C#).

> **NOTE**
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez Personnalisation de l'IDE.

Voir aussi

- Options du compilateur C#
- Désolé, nous n'avons pas d'informations sur cette erreur C#
- Générer, page du Concepteur de projets (C#)
- -Warn (options du compilateur C#)
- -nowarn (options du compilateur C#)

Présentation

02/11/2020 • 102 minutes to read • Edit Online

C# (prononcé « C Sharp ») est un langage de programmation simple, moderne, orienté objet et de type sécurisé. C#a ses racines dans la famille de langages C et sera immédiatement familière aux programmeurs C++Java, et Java. C#est standardisé par ECMA International comme norme ***ECMA-334*** et par ISO/IEC comme norme ***iso/IEC 23270*** . Le C# compilateur de Microsoft pour le .NET Framework est une implémentation conforme de ces deux normes.

C# est un langage orienté objet, mais C# inclut de plus la prise en charge de la programmation ***orientée composant***. La conception logicielle moderne s'appuie de plus en plus sur les composants logiciels sous la forme de packages de fonctionnalités autonomes et autodescriptifs. Point important, ces composants présentent un modèle de programmation avec propriétés, méthodes et événements ; ils ont des attributs qui fournissent des informations déclaratives sur le composant ; et ils intègrent leur propre documentation. C#fournit des constructions de langage pour prendre en charge directement ces C# concepts, en créant un langage très naturel dans lequel créer et utiliser des composants logiciels.

Plusieurs fonctionnalités de C# participent à la construction d'applications robustes et fiables : Le ***garbage collection*** libère automatiquement la mémoire occupée par les objets inutilisés ; la ***gestion des exceptions*** fournit une approche structurée et extensible de la détection et de la récupération des erreurs ; la conception de ***type sécurisé*** du langage rend impossible la lecture des variables non initialisées, l'indexation des tableaux au-delà de leurs limites ou l'exécution de conversions de type non vérifiées.

C# a un ***système de type unifié***. Tous les types C#, y compris les types primitifs tels que `int` et `double`, héritent d'un seul type `object` racine. Par conséquent, tous les types partagent un ensemble d'opérations communes, et des valeurs de tous types peuvent être stockées, transmises et exploitées de manière cohérente. En outre, C# prend en charge les types référence et les types valeur définis par l'utilisateur, ce qui permet l'allocation dynamique d'objets, ainsi que le stockage en ligne de structures légères.

Pour vous assurer C# que les programmes et les bibliothèques peuvent évoluer au fil du temps d'une manière compatible, l'accent est mis C#sur le contrôle de ***version*** dans la conception de. De nombreux langages de programmation n'accordent que peu d'attention à ce problème ; par conséquent, les programmes écrits dans ces langages s'arrêtent plus souvent que nécessaire lors de l'introduction de versions plus récentes des bibliothèques dépendantes. Les aspects C#de la conception de qui ont été directement influencés par les considérations `virtual` relatives `override` à la gestion des versions incluent les modificateurs séparés et, les règles de résolution de surcharge de méthode, ainsi que la prise en charge des déclarations de membres d'interface explicites.

Le reste de ce chapitre décrit les principales fonctionnalités du C# langage.

Bien que les chapitres ultérieurs décrivent les règles et les exceptions selon une méthode orientée détails et parfois mathématique, ce chapitre s'efforce de clarifier et de concision au détriment de l'exhaustivité. L'objectif est de fournir au lecteur une introduction au langage qui facilitera l'écriture de programmes précoces et la lecture des chapitres ultérieurs.

Hello World

Le programme « Hello, World » est souvent utilisé pour présenter un langage de programmation. Le voici en C# :

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

Les fichiers sources C# ont généralement l'extension de fichier `.cs`. En supposant que le programme « Hello, World » est stocké dans `hello.cs` le fichier, le programme peut être compilé avec C# le compilateur Microsoft à l'aide de la ligne de commande.

```
csc hello.cs
```

qui produit un assembly exécutable nommé `hello.exe`. La sortie produite par cette application lorsqu'elle est exécutée est

```
Hello, World
```

Le programme « Hello, World » commence par une directive `using` qui fait référence à l'espace de noms `System`. Les espaces de noms représentent un moyen hiérarchique d'organiser les bibliothèques et les programmes C#. Les espaces de noms contiennent des types et d'autres espaces de noms ; par exemple, l'espace de noms `System` contient plusieurs types, notamment la classe `Console` référencée dans le programme, et d'autres espaces de noms, tels que `IO` et `Collections`. Une directive `using` qui fait référence à un espace de noms donné permet l'utilisation non qualifiée des types membres de cet espace de noms. En raison de la directive `using`, le programme peut utiliser `Console.WriteLine` comme raccourci pour `System.Console.WriteLine`.

La classe `Hello` déclarée par le programme « Hello, World » a un membre unique, la méthode nommée `Main`. La `Main` méthode est déclarée avec `static` le modificateur. Si les méthodes d'instance peuvent faire référence à une instance d'objet englobante particulière avec le mot clé `this`, les méthodes statiques fonctionnent sans référence à un objet particulier. Par convention, une méthode statique nommée `Main` sert de point d'entrée d'un programme.

La sortie du programme est générée par la méthode `WriteLine` de la classe

`Console` dans l'espace de noms `System`. Cette classe est fournie par les bibliothèques de classes .NET Framework qui, par défaut, sont automatiquement référencées par le compilateur C# Microsoft. Notez qu' C# elle-même n'a pas de bibliothèque Runtime distincte. Au lieu de cela, le .NET Framework est la C#bibliothèque Runtime de.

Structure du programme

Les concepts clés d'organisation en C# sont les ***programmes***, ***espaces de noms***, ***types***, ***membres*** et ***assemblys***. Les programmes C++, comme les programmes C, se composent d'un ou plusieurs fichiers sources. Les programmes déclarent des types qui contiennent des membres et peuvent être organisés en espaces de noms. Les classes et les interfaces sont des exemples de types. Les champs, méthodes, propriétés et événements sont des exemples de membres. Lorsque les programmes C# sont compilés, ils sont physiquement empaquetés dans des assemblys. Les assemblys ont généralement l'extension `.exe` de `.dll` fichier ou, selon qu'ils implémentent des ***applications*** ou des ***bibliothèques***.

L'exemple

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

déclare une classe nommée `Stack` dans un espace de noms appelé. `Acme.Collections` Le nom qualifié complet de cette classe est `Acme.Collections.Stack`. La classe contient plusieurs membres : un champ nommé `top`, deux méthodes nommées `Push` et `Pop`, et une classe imbriquée nommée `Entry`. La classe `Entry` contient trois membres en plus : un champ nommé `next`, un autre nommé `data` et un constructeur.

En supposant que le code source de l'exemple est stocké dans le fichier `acme.cs`, la ligne de commande

```
csc /t:library acme.cs
```

compile l'exemple en tant que bibliothèque (code sans point d'entrée `Main`) et produit un assembly nommé `acme.dll`.

Les assemblys contiennent du code exécutable sous la forme d'instructions en ***langage intermédiaire*** (il) et des informations symboliques sous la forme de ***métadonnées***. Avant son exécution, le code de langage intermédiaire dans un assembly est automatiquement converti en code spécifique au processeur par le compilateur juste à temps (JIT) du Common Language Runtime .NET.

Comme un assembly est une unité de fonctionnalité autodescriptive contenant du code et des métadonnées, les directives `#include` et les fichiers d'en-tête ne sont pas nécessaires en C#. Les membres et types publics contenus dans un assembly particulier sont disponibles dans un programme C# par simple référence à cet assembly lors de la compilation du programme. Par exemple, ce programme utilise la classe `Acme.Collections.Stack` à partir de l'assembly `acme.dll` :

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

Si le programme est stocké dans le `test.cs` fichier, `test.cs` lorsque est compilé, `acme.dll` l'assembly peut être référencé à l'aide `/r` de l'option du compilateur :

```
csc /r:acme.dll test.cs
```

Cela permet de créer un assembly exécutable nommé `test.exe`, qui, lors de l'exécution, produit la sortie :

```
100
10
1
```

C# permet le stockage du texte source d'un programme dans plusieurs fichiers source. Lorsqu'un programme C# multifichier est compilé, tous les fichiers source sont traités ensemble et les fichiers source peuvent

librement se référencer mutuellement. Sur le plan conceptuel, c'est comme si tous les fichiers source étaient concaténés en un seul fichier volumineux avant d'être traités. Déclarations anticipées ne sont jamais nécessaires en C#, car, à de très rares exceptions près, l'ordre de déclaration n'a pas d'importance. C# ne limite pas un fichier source à la déclaration d'un seul type public et ne nécessite pas non plus que le nom du fichier source corresponde à un type déclaré dans ce fichier.

Types et variables

Il existe deux genres de types en C# : les ***types référence*** et les ***types valeur***. Les variables des types valeur contiennent directement leurs données alors que les variables des types référence contiennent des références à leurs données, connues sous le nom d'objets. Avec les types référence, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, les variables possèdent leur propre copie de données, et les opérations sur une variable ne peuvent absolument pas affecter l'autre (sauf pour les variables de paramètre `ref` et `out`).

C#les types valeur de sont encore divisés en ***types simples***, ***types ENUM***, ***types struct***et ***types Nullable***, C#et les types référence de sont encore divisés en ***types classe***, ***types interface***, ***tableau types***et ***types délégués***.

Le tableau suivant fournit une vue d' C#ensemble du système de type de.

CATÉGORIE		DESCRIPTION
Types de valeur	Types simples	Entier signé : `sbyte`, `short`, `int`, `long`
		Entier non signé : `byte`, `ushort`, `uint`, `ulong`
		Caractères Unicode : `char`
		Virgule flottante IEEE : `float`, `double`
		Décimale haute précision : `decimal`
		Booléen : `bool`
	Types d'enum	Types définis par l'utilisateur de la forme `enum E {...}`
	Types struct	Types définis par l'utilisateur de la forme `struct S {...}`

CATÉGORIE		DESCRIPTION
	Types Nullable	Extensions de tous les autres types de valeurs avec une valeur `null`
Types référence	Types de classes	Classe de base fondamentale de tous les autres types : `object`
		Chaînes Unicode : `string`
		Types définis par l'utilisateur de la forme `class C {...}`
	Types interface	Types définis par l'utilisateur de la forme `interface I {...}`
	Types de tableaux	Uni et multidimensionnels, par exemple `int[]` et `int[,]`
	Types délégués	Les types définis par l'utilisateur de la forme, par exemple `delegate int D(...)`

Les types intégraux huit prennent en charge les valeurs 8 bits, 16 bits, 32 bits et 64 bits sous forme signée ou non signée.

Les deux types à virgule flottante `double`, `float` et, sont représentés à l'aide des formats IEEE 754 simple précision et double précision 64 bits 32.

Le type `decimal` est un type de données 128 bits adapté aux calculs financiers et monétaires.

C#le `bool` type de est utilisé pour représenter des valeurs booléennes (valeurs `true` qui `false` sont ou).

Le traitement des caractères et chaînes dans le langage C# utilise l'encodage Unicode. Le type `char` représente une unité de code UTF-16, et le type `string` représente une séquence d'unités de code UTF-16.

Le tableau suivant récapitule les C#types numériques de.

CATÉGORIE	BITS	TYPE	PLAGE/PRÉCISION
Entier signé	8	`sbyte`	-128... 127
	16	`short`	-32768... 32, 767

CATÉGORIE	BITS	TYPE	PLAGE/PRÉCISION
	32	`int`	-2147483648… 2, 147, 483, 647
	64	`long`	-9223372036854775808… 9, 223, 372, 036, 854, 775, 807
Entier non signé	8	`byte`	0… 255
	16	`ushort`	0… 65, 535
	32	`uint`	0… 4, 294, 967, 295
	64	`ulong`	0… 18, 446, 744, 073, 709, 551, 615
Virgule flottante	32	`float`	1,5 × 10 ^ − 45 à 3,4 × 10 ^ 38, précision à 7 chiffres
	64	`double`	5,0 × 10 ^ − 324 à 1,7 × 10 ^ 308, précision de 15 chiffres
Decimal	128	`decimal`	1,0 × 10 ^ − 28 à 7,9 × 10 ^ 28, précision de 28 chiffres

Les programmes C# utilisent les ***déclarations de type*** pour créer de nouveaux types. Une déclaration de type spécifie le nom et les membres du nouveau type. Cinq des C#catégories de types de sont définissables par l'utilisateur : types de classes, types struct, types interface, types ENUM et types délégués.

Un type de classe définit une structure de données qui contient des données membres (champs) et des fonctions membres (méthodes, propriétés et autres). Les types de classes prennent en charge l'héritage unique et le polymorphisme, des mécanismes par lesquels les classes dérivées peuvent étendre et spécialiser les classes de base.

Un type struct est semblable à un type de classe, car il représente une structure avec des membres de données et des fonctions membres. Toutefois, contrairement aux classes, les structs sont des types valeur et ne nécessitent pas d'allocation de tas. Les types struct ne prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object`.

Un type interface définit un contrat comme un jeu nommé de membres de fonction publics. Une classe ou un struct qui implémente une interface doit fournir des implémentations des membres de la fonction de l'interface. Une interface peut hériter de plusieurs interfaces de base, et une classe ou un struct peut implémenter plusieurs interfaces.

Un type délégué représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction dans d'autres langages, mais contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

Les types de classe, de struct, d'interface et de délégué prennent tous en charge les génériques, dans lesquels ils peuvent être paramétrés avec d'autres types.

Un type enum est un type distinct avec des constantes nommées. Chaque type enum a un type sous-jacent, qui doit être l'un des huit types intégraux. L'ensemble de valeurs d'un type enum est le même que l'ensemble de valeurs du type sous-jacent.

C# prend en charge les tableaux uni et multidimensionnels de tout type. Contrairement aux types mentionnés ci-dessus, les types de tableaux n'ont pas à être déclarés avant de pouvoir être utilisés. Au lieu de cela, les types de tableaux sont construits en ajoutant des crochets à un nom de type. Par exemple, `int[]` est un tableau unidimensionnel de `int`, `int[,]` est un tableau à deux dimensions de `int`, et `int[][]` est un tableau unidimensionnel de tableaux unidimensionnels de `int`.

Les types Nullable n'ont pas besoin d'être déclarés avant de pouvoir être utilisés. Pour chaque type `T` valeur n'acceptant pas les valeurs NULL, il `T?` existe un type Nullable correspondant qui peut `null` contenir une valeur supplémentaire. Par exemple, `int?` est un type qui peut contenir n'importe quel entier 32 bits ou `null` la valeur.

C#le système de type de est unifié de telle sorte qu'une valeur de n'importe quel type peut être traitée en tant qu'objet. Chaque type dans C# dérive directement ou indirectement du type `object`, et `object` est la classe de base fondamentale de tous les types. Les valeurs des types référence sont considérées comme des objets simplement en affichant les valeurs en tant que type `object`. Les valeurs des types valeur sont traitées comme des objets en effectuant des opérations de ***boxing*** et d' ***unboxing*** . Dans l'exemple suivant, une valeur `int` est convertie en `object` et à nouveau en `int`.

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;          // Boxing
        int j = (int)o;        // Unboxing
    }
}
```

Quand une valeur d'un type valeur est convertie `object` en type, une instance d'objet, également appelée « Box », est allouée pour contenir la valeur, et la valeur est copiée dans cette zone. À l'inverse, lorsqu'une `object` référence est castée en un type valeur, une vérification est effectuée que l'objet référencé est une zone du type valeur correct et, si la vérification est réussie, la valeur dans la zone est copiée.

C#le système de type unifié de est efficace signifie que les types valeur peuvent devenir des objets « à la demande ». En raison de l'unification, les bibliothèques à usage général qui utilisent le type `object` peuvent être utilisées avec les types référence et les types valeur.

Il existe plusieurs types de ***variables*** en C#, y compris les champs, les éléments de tableau, les variables locales et les paramètres. Les variables représentent des emplacements de stockage, et chaque variable a un type qui détermine les valeurs qui peuvent être stockées dans la variable, comme indiqué dans le tableau suivant.

TYPE DE VARIABLE	CONTENU POSSIBLE
Type de valeur n'acceptant pas Null	Une valeur de ce type exact
Types valeur Nullable	Une valeur null ou une valeur de ce type exact
`object`	Une référence null, une référence à un objet de n'importe quel type référence ou une référence à une valeur boxed de n'importe quel type valeur
Type de classe	Une référence null, une référence à une instance de ce type de classe ou une référence à une instance d'une classe dérivée de ce type de classe
Type d'interface	Une référence null, une référence à une instance d'un type de classe qui implémente ce type d'interface, ou une référence à une valeur boxed d'un type valeur qui implémente ce type d'interface
Type tableau	Une référence null, une référence à une instance de ce type de tableau ou une référence à une instance d'un type de tableau compatible

TYPE DE VARIABLE	CONTENU POSSIBLE
Type délégué	Une référence null ou une référence à une instance de ce type délégué

Expressions

Les ***expressions*** sont construites à partir de ***d'opérandes*** et ***d'opérateurs***. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes. Parmi les exemples d'opérateurs figurent `+`, `-`, `*`, `/` et `new`. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Quand une expression contient plusieurs opérateurs, la ***priorité*** des opérateurs contrôle l'ordre dans lequel les opérateurs individuels sont évalués. Par exemple, l'expression `x + y * z` est évaluée comme `x + (y * z)`, car l'opérateur `*` a une priorité plus élevée que `+`.

La plupart des opérateurs peuvent être ***surchargés***. La surcharge d'opérateur autorise la spécification d'implémentations d'opérateurs définies par l'utilisateur pour les opérations où l'un des deux opérandes ou les deux sont d'un type classe ou struct défini par l'utilisateur.

Le tableau suivant récapitule les C#opérateurs de, en répertoriant les catégories d'opérateurs par ordre de priorité, de la plus élevée à la plus faible. Les opérateurs de la même catégorie ont la même priorité.

CATÉGORIE	EXPRESSION	DESCRIPTION
Principale	`x.m`	Accès au membre
	`x(...)`	Méthode et appel de délégué
	`x[...]`	Tableau et accès d'indexeur
	`x++`	Post-incrémentation
	`x--`	Post-décrémentation
	`new T(...)`	Création d'objet et de délégué
	`new T(...){...}`	Création d'objet avec initialiseur
	`new {...}`	Initialiseur d'objet anonyme
	`new T[...]`	Création de tableau
	`typeof(T)`	Obtenir l'objet `System.Type` de `T`

CATÉGORIE	EXPRESSION	DESCRIPTION
	`checked(x)`	Évaluer l'expression dans le contexte vérifié (checked)
	`unchecked(x)`	Évaluer l'expression dans le contexte non vérifié (unchecked)
	`default(T)`	Obtenir la valeur par défaut du type `T`
	`delegate {...}`	Fonction anonyme (méthode anonyme)
Unaire	`+x`	Identité
	`-x`	Négation
	`!x`	Négation logique
	`~x`	Négation d'opération de bits
	`++x`	Pré-incrémentation
	`--x`	Pré-décrémentation
	`(T)x`	Convertir explicitement `x` en type `T`
	`await x`	Attendre de façon asynchrone la fin de `x`
Multiplication	`x * y`	Multiplication
	`x / y`	Division
	`x % y`	Reste
Addition	`x + y`	Addition, concaténation de chaînes, combinaison de délégués
	`x - y`	Soustraction, suppression de délégué
Shift	`x << y`	Décalage à gauche
	`x >> y`	Décalage à droite
Relations et test de type	`x < y`	Inférieur à

CATÉGORIE	EXPRESSION	DESCRIPTION
	`x > y`	Supérieur à
	`x <= y`	Inférieur ou égal à
	`x >= y`	Supérieur ou égal à
	`x is T`	Retourne `true` si `x` est un `T`, `false` sinon
	`x as T`	Retourne `x` de type `T`, ou `null` si `x` n'est pas un `T`
Égalité	`x == y`	Égal à
	`x != y`	Non égal à
AND logique	`x & y`	AND d'entiers au niveau du bit, AND logique booléen
XOR logique	`x ^ y`	Opération de bits entière XOR, Boolean logique XOR
OR logique	`x | y`	Opération de bits entière OR, Boolean logique OR
AND conditionnel	`x && y`	`x` Évalue `y` uniquement si est `true`
OR conditionnel	`x || y`	`x` Évalue `y` uniquement si est `false`
Fusion de Null	`x ?? y`	Prend la valeur `x` `null` si est , `x` sinon `y`
Conditionnel	`x ? y : z`	Prend la valeur `y` si `x` est `true`, `z` si `x` est `false`
Attribution ou fonction anonyme	`x = y`	Attribution
	`x op= y`	Assignation composée ; les opérateurs pris `*=` en charge sont `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`

CATÉGORIE	EXPRESSION	DESCRIPTION
	`(T x) => y`	Fonction anonyme (expression lambda)

Instructions

Les actions d'un programme sont exprimées à l'aide ***d'instructions***. C# prend en charge plusieurs types d'instructions, dont plusieurs sont définies en termes d'instructions incorporées.

Un ***bloc*** autorise l'écriture de plusieurs instructions dans les contextes où une seule instruction est autorisée. Un bloc se compose d'une liste d'instructions écrites entre les délimiteurs `{` et `}`.

Les ***instructions de déclaration*** sont utilisées pour déclarer des variables locales et des constantes.

Les ***instructions d'expression*** sont utilisées pour évaluer des expressions. Les expressions qui peuvent être utilisées en tant qu'instructions incluent les appels de méthode, les allocations d'objets à `=` l'aide de l' `new` opérateur, les assignations utilisant et les opérateurs d'assignation composée, les opérations d'incrémentation et de décrémentation à l'aide de `++` opérateurs `--` et et expressions await.

Les ***instructions de sélection*** sont utilisées pour sélectionner un nombre d'instructions possibles pour l'exécution en fonction de la valeur d'une expression. Dans ce groupe, ce sont les instructions `if` et `switch`.

Les ***instructions d'itération*** sont utilisées pour exécuter une instruction incorporée de façon répétée. Dans ce groupe, ce sont les instructions `while`, `do`, `for` et `foreach`.

Les ***instructions de saut*** sont utilisées pour transférer le contrôle. Dans ce groupe, ce sont les instructions `break`, `continue`, `goto`, `throw`, `return` et `yield`.

L'instruction `try` ... `catch` est utilisée pour intercepter les exceptions qui se produisent pendant l'exécution d'un bloc, et l'instruction `try` ... `finally` permet de spécifier que le code de finalisation est toujours exécuté, qu'une exception se soit produite ou non.

Les `checked` instructions `unchecked` et sont utilisées pour contrôler le contexte de contrôle de dépassement de capacité pour les opérations arithmétiques de type intégral et les conversions.

L'instruction `lock` est utilisée pour obtenir le verrou d'exclusion mutuelle pour un objet donné, exécuter une instruction, puis libérer le verrou.

L'instruction `using` est utilisée pour obtenir une ressource, exécuter une instruction puis supprimer cette ressource.

Voici des exemples de chaque type d'instruction

Déclarations de variables locales

```
static void Main() {
   int a;
   int b = 2, c = 3;
   a = 1;
   Console.WriteLine(a + b + c);
}
```

Déclaration de constante locale

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Instruction d'expression

```
static void Main() {
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

instruction `if`

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

instruction `switch`

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

instruction `while`

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

instruction `do`

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

instruction `for`

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

instruction `foreach`

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

instruction `break`

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

instruction `continue`

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

instruction `goto`

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

instruction `return`

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

instruction `yield`

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` instructions `try` et

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` instructions `unchecked` et

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

instruction `lock`

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

instruction `using`

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Classes et objets

Les ***classes*** représentent le type le plus fondamental de C#. Une classe est une structure de données qui combine l'état (champs) et les actions (méthodes et autres fonctions membres) en une seule unité. Une classe fournit une définition pour les ***instances*** créées dynamiquement de la classe, également appelées ***objets***. Les classes prennent en charge ***l'héritage*** et le ***polymorphisme***, des mécanismes par lesquels les ***classes dérivées*** peuvent étendre et spécialiser les ***classes de base***.

Les nouvelles classes sont créées à l'aide des déclarations de classe. Une déclaration de classe commence par un en-tête qui spécifie les attributs et modificateurs de la classe, le nom de la classe, la classe de base (si fournie) et les interfaces implémentées par la classe. L'en-tête est suivi par le corps de la classe, qui se compose d'une liste de déclarations de membres écrites entre les délimiteurs `{` et `}`.

Voici une déclaration d'une classe simple nommée `Point` :

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Les instances de classes sont créées à l'aide de l'opérateur `new`, qui alloue la mémoire pour une nouvelle instance, appelle un constructeur pour initialiser l'instance et retourne une référence à l'instance. Les instructions suivantes créent deux `Point` objets et stockent les références à ces objets dans deux variables :

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La mémoire occupée par un objet est automatiquement récupérée lorsque l'objet n'est plus utilisé. Il n'est ni possible ni nécessaire de libérer explicitement des objets dans C#.

Membres

Les membres d'une classe sont soit des ***membres statiques*** , soit des ***membres d'instance***. Les membres statiques appartiennent à des classes, et les membres d'instance appartiennent à des objets (instances de classes).

Le tableau suivant fournit une vue d'ensemble des types de membres qu'une classe peut contenir.

MEMBRE	DESCRIPTION
Constantes	Valeurs constantes associées à la classe
Champs	Variables de la classe
Méthodes	Calculs et les actions qui peuvent être effectués par la classe
Propriétés	Actions associées à la lecture et l'écriture des propriétés nommées de la classe
Indexeurs	Actions liées à l'indexation des instances de la classe comme un tableau
Événements	Les notifications qui peuvent être générées par la classe
Opérateurs	Les opérateurs de conversion et d'expression pris en charge par la classe

MEMBRE	DESCRIPTION
Constructeurs	Les actions requises pour initialiser les instances de la classe ou la classe elle-même
Destructeurs	Actions à effectuer avant que les instances de la classe soient abandonnées de façon définitive
Types	Types imbriqués déclarés par la classe

Accessibilité

Chaque membre d'une classe a une accessibilité associée, qui contrôle les régions du texte du programme qui sont en mesure d'accéder au membre. Il existe cinq formes possibles d'accessibilité. Ceux-ci sont résumés dans le tableau ci-après.

ACCESSIBILITÉ	SIGNIFICATION
`public`	Accès non limité
`protected`	Accès limité à cette classe ou aux classes dérivées de cette classe
`internal`	Accès limité à ce programme
`protected internal`	Accès limité à ce programme ou aux classes dérivées de cette classe
`private`	Accès limité à cette classe

Paramètres de type

Une définition de classe peut spécifier un jeu de paramètres de type en faisant suivre le nom de classe par une liste de noms de paramètre de type entre crochets. Les paramètres de type peuvent être utilisés dans le corps des déclarations de classe pour définir les membres de la classe. Dans l'exemple suivant, les paramètres de type de `Pair` sont `TFirst` et `TSecond` :

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Un type de classe déclaré pour prendre des paramètres de type est appelé un type de classe générique. Les types struct, interface et délégué peuvent également être génériques.

Lorsque la classe générique est utilisée, des arguments de type doivent être fournis pour chacun des paramètres de type :

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two"
};
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Un type générique avec des arguments de type fournis `Pair<int,string>`, comme ci-dessus, est appelé un type construit.

Classes de base

Une déclaration de classe peut spécifier une classe de base en faisant suivre les paramètres de nom et de type de classe par un signe deux-points et le nom de la classe de base. L'omission d'une spécification de classe de base revient à dériver du type `object`. Dans l'exemple suivant, la classe de base de `Point3D` est `Point`, et la classe de base de `Point` est `object` :

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

Une classe hérite des membres de sa classe de base. L'héritage signifie qu'une classe contient implicitement tous les membres de sa classe de base, à l'exception de l'instance et des constructeurs statiques, et des destructeurs de la classe de base. Une classe dérivée peut ajouter des membres hérités, mais ne peut pas supprimer la définition d'un membre hérité. Dans l'exemple précédent, `Point3D` hérite des champs `x` et `y` de `Point`, et chaque instance de `Point3D` contient trois champs, `x`, `y` et `z`.

Il existe une conversion implicite d'un type de classe vers un de ses types de classe de base. Par conséquent, une variable d'un type de classe peut référencer une instance de cette classe ou une instance d'une classe dérivée. Par exemple, pour les déclarations de classe précédentes, une variable de type `Point` peut faire référence à un objet `Point` ou `Point3D` :

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Champs

Un champ est une variable associée à une classe ou à une instance d'une classe.

Un champ déclaré avec le `static` modificateur définit un ***champ statique***.

Un champ statique identifie exactement un seul emplacement de stockage. Quel que soit le nombre d'instances d'une classe qui sont créées, il n'existe qu'une seule copie d'un champ statique.

Un champ déclaré sans le `static` modificateur définit un ***champ d'instance***. Chaque instance d'une classe contient une copie distincte de tous les champs d'instance de cette classe.

Dans l'exemple suivant, chaque instance de la classe `Color` possède une copie distincte des champs d'instance `r`, `g` et `b`, mais il existe une seule copie des champs statiques `Black`, `White`, `Red`, `Green` et `Blue` :

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Comme indiqué dans l'exemple précédent, les ***champs en lecture seule*** peuvent être déclarés avec un modificateur `readonly`. L'assignation à `readonly` un champ ne peut se produire que dans le cadre de la déclaration du champ ou dans un constructeur de la même classe.

Méthodes

Une ***méthode*** est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les ***méthodes statiques*** sont accessibles à travers la classe. Les ***méthodes d'instance*** sont accessibles via des instances de la classe.

Les méthodes ont une liste (éventuellement vide) de ***paramètres***, qui représentent des valeurs ou des références variables passées à la méthode, et un ***type de retour***, qui spécifie le type de la valeur calculée et retournée par la méthode. Le type de retour d'une `void` méthode est s'il ne retourne pas de valeur.

Comme les types, les méthodes peuvent également être un jeu de paramètres de type pour lesquels les arguments de type doivent être spécifiés lorsque la méthode est appelée. Contrairement aux types, les arguments de type peuvent souvent être déduits à partir des arguments d'un appel de méthode et n'ont pas à être fournis explicitement.

La ***signature*** d'une méthode doit être unique dans la classe dans laquelle la méthode est déclarée. La signature d'une méthode se compose du nom de la méthode, du nombre de paramètres de type et du nombre, des modificateurs et des types de ses paramètres. La signature d'une méthode n'inclut pas le type de retour.

Paramètres

Les paramètres sont utilisés pour passer des valeurs ou des références variables aux méthodes. Les paramètres d'une méthode obtiennent leurs valeurs réelles à partir des ***arguments*** qui sont spécifiés lorsque la méthode est appelée. Il existe quatre types de paramètres : les paramètres de valeur, les paramètres de référence, les paramètres de sortie et les tableaux de paramètres.

Un ***paramètre de valeur*** est utilisé pour le passage de paramètres d'entrée. Un paramètre de valeur correspond à une variable locale qui obtient sa valeur initiale de l'argument qui a été transmis pour le paramètre. Les modifications apportées à un paramètre de valeur n'affectent pas l'argument qui a été transmis pour le paramètre.

Les paramètres de valeur peuvent être facultatifs, en spécifiant une valeur par défaut afin que les arguments correspondants puissent être omis.

Un ***paramètre de référence*** est utilisé pour la transmission de paramètres en entrée et en sortie. L'argument passé pour un paramètre de référence doit être une variable, et pendant l'exécution de la méthode, le paramètre de référence représente le même emplacement de stockage que la variable d'argument. Un paramètre de référence est déclaré avec le modificateur `ref`. L'exemple suivant illustre l'utilisation des paramètres `ref`.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);            // Outputs "2 1"
    }
}
```

Un ***paramètre de sortie*** est utilisé pour le passage de paramètres de sortie. Un paramètre de sortie est similaire à un paramètre de référence, sauf que la valeur initiale de l'argument fourni par l'appelant n'a pas d'importance. Un paramètre de sortie est déclaré avec le modificateur `out`. L'exemple suivant illustre l'utilisation des paramètres `out`.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder)
{
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

Un ***tableau de paramètres*** autorise le passage d'un nombre variable d'arguments à une méthode. Un tableau de paramètres est déclaré avec le modificateur `params`. Seul le dernier paramètre d'une méthode peut être un tableau de paramètres, et le type d'un tableau de paramètres doit être un type tableau unidimensionnel. Les `Write` méthodes `WriteLine` et de la `System.Console` classe sont de bons exemples d'utilisation des tableaux de paramètres. Vous les déclarez de la façon suivante.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Dans une méthode qui utilise un tableau de paramètres, le tableau de paramètres se comporte exactement comme un paramètre ordinaire de type tableau. Toutefois, dans un appel à une méthode avec un tableau de paramètres, il est possible de passer un argument unique de type tableau de paramètres ou n'importe quel nombre d'arguments du type d'élément du tableau de paramètres. Dans ce cas, une instance de tableau est automatiquement créée et initialisée avec les arguments donnés. Cet exemple

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

revient à écrire ce qui suit.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Corps de la méthode et variables locales

Le corps d'une méthode spécifie les instructions à exécuter lorsque la méthode est appelée.

Un corps de méthode peut déclarer des variables qui sont spécifiques à l'appel de la méthode. Ces variables sont appelées ***variables locales***. Une déclaration de variable locale spécifie un nom de type, un nom de variable et éventuellement une valeur initiale. L'exemple suivant déclare une variable locale `i` avec une valeur initiale de zéro et une variable locale `j` sans valeur initiale.

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requiert qu'une variable locale soit ***assignée de manière définitive*** avant de pouvoir obtenir sa valeur. Par exemple, si la déclaration du `i` précédent n'inclut pas de valeur initiale, le compilateur signale une erreur pour les utilisations ultérieures de `i`, car `i` ne serait pas assigné de manière définitive à ces points dans le programme.

Une méthode peut utiliser les instructions `return` pour retourner le contrôle à son appelant. Dans une méthode retournant `void`, les instructions `return` ne peuvent pas spécifier une expression. Dans une méthode qui retourne des `void` instructions `return` non-, doit inclure une expression qui calcule la valeur de retour.

Méthodes statiques et d'instance

Une méthode déclarée avec `static` un modificateur est une ***méthode statique***. Une méthode statique n'opère pas sur une instance spécifique et permet uniquement d'accéder directement à des membres statiques.

Une méthode déclarée sans `static` modificateur est une ***méthode d'instance***. Une méthode d'instance opère sur une instance spécifique et peut accéder aux membres statiques et d'instance. L'instance sur laquelle une méthode d'instance a été appelée est explicitement accessible en tant que `this`. Une erreur consiste à faire référence à `this` dans une méthode statique.

La classe `Entity` suivante a à la fois des statiques et des membres d'instance.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

Chaque instance `Entity` contient un numéro de série (et probablement d'autres informations qui ne sont pas indiquées ici). Le constructeur `Entity` (qui est similaire à une méthode d'instance) initialise la nouvelle instance avec le numéro de série suivant. Étant donné que le constructeur est un membre d'instance, il est autorisé à accéder à la fois au champ d'instance `serialNo` et au champ statique `nextSerialNo`.

Les méthodes statiques `GetNextSerialNo` et `SetNextSerialNo` peuvent accéder au champ statique `nextSerialNo`, mais ce serait une erreur pour eux d'accéder directement au champ d'instance `serialNo`.

L'exemple suivant illustre l'utilisation de la `Entity` classe.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Notez que les méthodes statiques `SetNextSerialNo` et `GetNextSerialNo` sont appelées sur la classe alors que la méthode d'instance `GetSerialNo` est appelée sur les instances de la classe.

Méthodes virtuelles, de substitution et abstraites

Lorsqu'une déclaration de méthode d'instance inclut un modificateur `virtual`, la méthode est appelée ***méthode virtuelle***. Quand aucun `virtual` modificateur n'est présent, la méthode est dite ***non virtuelle***.

Lorsqu'une méthode virtuelle est appelée, le ***type au moment de l'exécution*** de l'instance pour laquelle cet appel prend place détermine

l'implémentation de méthode à appeler. Dans un appel de méthode non virtuelle, le ***type lors de la compilation*** de l'instance est le facteur déterminant.

Une méthode virtuelle peut être ***substituée*** dans une classe dérivée. Lorsqu'une déclaration de méthode d'instance `override` comprend un modificateur, la méthode substitue une méthode virtuelle héritée avec la même signature. Là où une déclaration de méthode virtuelle présente une nouvelle méthode, une déclaration de méthode de substitution spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

Une méthode ***abstraite*** est une méthode virtuelle sans implémentation. Une méthode abstraite est déclarée `abstract` avec le modificateur et est autorisée uniquement dans une classe qui est `abstract` également déclarée. Une méthode abstraite doit être remplacée dans chaque classe dérivée non abstraite.

L'exemple suivant déclare une classe abstraite, `Expression`, qui représente un nœud d'arborescence de l'expression et trois classes dérivées, `Constant`, `VariableReference` et `Operation`, qui implémentent des nœuds d'arborescence de l'expression pour les références variables, les constantes et les opérations arithmétiques. (Cela est similaire à, mais à ne pas confondre avec les types d'arborescence d'expression introduits dans les types d'arborescence d'expression).

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

Les quatre classes précédentes permet de modéliser des expressions arithmétiques. Par exemple, en utilisant des instances de ces classes, l'expression `x + 3` peut être représentée comme suit.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

La méthode `Evaluate` d'une instance `Expression` est appelée pour évaluer l'expression donnée et produire une valeur `double`. La méthode prend comme argument un `Hashtable` qui contient des noms de variables (comme clés des entrées) et des valeurs (comme valeurs des entrées). La `Evaluate` méthode est une méthode abstraite virtuelle, ce qui signifie que les classes dérivées non abstraites doivent être substituées pour fournir une implémentation réelle.

Une implémentation de `Constant` de `Evaluate` renvoie simplement la constante stockée. L' `VariableReference` implémentation de A recherche le nom de variable dans la Hashtable et retourne la valeur résultante. Une implémentation de `Operation` évalue d'abord les opérandes de gauche et de droite (en appelant de manière récursive leurs méthodes `Evaluate`), puis effectue l'opération arithmétique donnée.

Le programme suivant utilise les classes `Expression` pour évaluer l'expression `x * (y + 2)` pour différentes valeurs de `x` et `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

Surcharge de méthode

La ***surcharge*** de méthode permet d'avoir plusieurs méthodes dans la même classe avec le même nom, tant qu'elles ont des signatures uniques. Lors de la compilation d'un appel à une méthode surchargée, le compilateur utilise ***la résolution de surcharge*** pour déterminer la méthode spécifique à appeler. La résolution de surcharge trouve la méthode qui correspond le mieux aux arguments ou signale une erreur si aucune meilleure correspondance ne peut être trouvée. L'exemple suivant montre la résolution de surcharge en action. Le commentaire pour chaque

appel de la méthode `Main` affiche une méthode qui est réellement appelée.

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

Comme le montre l'exemple, une méthode particulière peut toujours être sélectionnée en effectuant un cast explicite des arguments aux types de paramètres exacts et/ou en fournissant explicitement les arguments de type.

Autres fonctions membres

Les membres qui contiennent du code exécutable sont collectivement regroupés sous les ***membres de fonction*** d'une classe. La section précédente décrit les méthodes qui sont du type principal des fonctions membres. Cette section décrit les autres genres de fonctions membres pris en C#charge par : les constructeurs, les propriétés, les indexeurs, les événements, les opérateurs et les destructeurs.

Le code suivant illustre une classe générique appelée `List<T>`, qui implémente une liste d'objets pouvant être augmentée. La classe contient plusieurs exemples des types les plus courants de membres de fonction.

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
```

```
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
```

```
        return !Equals(a, b);
    }
}
```

Constructeurs

C# prend en charge les constructeurs statiques et d'instance. Un ***constructeur d'instance*** est un membre qui implémente les actions requises pour initialiser une instance d'une classe. Un ***constructeur statique*** est un membre qui implémente les actions requises pour initialiser une classe lui-même lorsqu'il est chargé.

Un constructeur est déclaré comme une méthode sans aucun type de retour et le même nom que la classe contenante. Si une déclaration de constructeur comprend `static` un modificateur, elle déclare un constructeur statique. Dans le cas contraire, elle déclare un constructeur d'instance.

Les constructeurs d'instance peuvent être surchargés. Par exemple, la classe `List<T>` déclare deux constructeurs d'instance, un sans paramètres et une fonction qui accepte un paramètre `int`. Les constructeurs d'instance sont appelés en utilisant l'opérateur `new`. Les instructions suivantes allouent `List<string>` deux instances à l'aide de chacun des constructeurs `List` de la classe.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Contrairement aux autres membres, les constructeurs d'instance ne sont pas hérités, et une classe n'a aucun constructeur d'instance autre que ceux réellement déclarés dans la classe. Si aucun constructeur d'instance n'est fourni pour une classe, un constructeur vide sans paramètre est fourni automatiquement.

Propriétés

Les ***propriétés*** sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés, et la syntaxe pour accéder aux champs et propriétés est la même. Toutefois, contrairement aux champs, les propriétés ne désignent pas des emplacements de stockage. Au lieu de cela, les propriétés ont des ***accesseurs*** qui spécifient les instructions à exécuter lorsque les valeurs sont lues ou écrites.

Une propriété est déclarée comme un champ `get` , sauf que la déclaration se termine par un accesseur et/ou un `set` accesseur écrit `{` entre `}` les délimiteurs et au lieu de se terminer par un point-virgule. Une propriété qui a à la `get` fois un accesseur et un `set` accesseur est une propriété ***en lecture-écriture***, une `get` propriété qui a uniquement un accesseur est une ***propriété en lecture seule***, et `set` une propriété qui a uniquement un accesseur est un ***propriété en écriture seule***.

Un `get` accesseur correspond à une méthode sans paramètre avec une valeur de retour du type de propriété. À l'exception de la cible d'une assignation, lorsqu'une propriété est référencée dans une expression `get` , l'accesseur de la propriété est appelé pour calculer la valeur de la propriété.

Un `set` accesseur correspond à une méthode avec un paramètre unique `value` nommé et aucun type de retour. Lorsqu'une propriété est référencée comme cible d'une assignation ou comme opérande de `++` ou `--`, l' `set` accesseur est appelé avec un argument qui fournit la nouvelle valeur.

La classe `List<T>` déclare deux propriétés, `Count` et `Capacity`, qui sont respectivement en lecture seule et en lecture-écriture. Voici un exemple d'utilisation de ces propriétés.

```
List<string> names = new List<string>();
names.Capacity = 100;            // Invokes set accessor
int i = names.Count;             // Invokes get accessor
int j = names.Capacity;          // Invokes get accessor
```

C# prend en charge les propriétés d'instance et les propriétés statiques, similaires aux champs et aux méthodes. Les propriétés statiques sont déclarées avec le modificateur, et les `static` propriétés d'instance sont déclarées sans lui.

Le ou les accesseurs d'une propriété peuvent être virtuels. Lorsqu'une déclaration de propriété inclut un modificateur `virtual`, `abstract` ou `override`, elle s'applique aux accesseurs de la propriété.

Indexeurs

Un ***indexeur*** est un membre qui permet l'indexation des objets de la même façon en tant que tableau. Un indexeur est déclaré comme une propriété, sauf que le nom du membre est `this`, suivi d'une liste de paramètres écrits entre les délimiteurs `[` et `]`. Les paramètres sont disponibles dans le ou les accesseurs de l'indexeur. Similaires aux propriétés, les indexeurs peuvent être en lecture-écriture, en lecture seule et en écriture seule, et les accesseurs d'un indexeur peuvent être virtuels.

La classe `List` déclare un indexeur en lecture-écriture unique qui prend un paramètre `int`. L'indexeur rend possible l'indexation des instances `List` avec des valeurs `int`. Exemple :

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Les indexeurs peuvent être surchargés, ce qui signifie qu'une classe peut déclarer plusieurs indexeurs tant que le nombre ou les types de leurs paramètres diffèrent.

Événements

Un ***événement*** est un membre qui permet à une classe ou un objet de fournir des notifications. Un événement est déclaré comme un champ, sauf que la déclaration comprend `event` un mot clé et que le type doit être un type délégué.

Dans une classe qui déclare un membre d'événement, l'événement se

comporte comme un champ d'un type délégué (à condition que l'événement n'est pas abstrait et ne déclare pas d'accesseurs). Le champ stocke une référence à un délégué qui représente les gestionnaires d'événements qui ont été ajoutés à l'événement. Si aucun descripteur d'événement n'est présent `null`, le champ est.

La classe `List<T>` déclare un membre d'événement unique appelé `Changed`, qui indique qu'un nouvel élément a été ajouté à la liste. L' `Changed` événement est déclenché par la `OnChanged` méthode virtuelle, qui vérifie d'abord si l'événement `null` est (ce qui signifie qu'aucun gestionnaire n'est présent). La notion de déclenchement d'un événement est équivalente à l'appel de délégué représenté par l'événement. Par conséquent, il n'existe aucune construction de langage particulière pour déclencher des événements.

Les clients réagissent aux événements via les ***gestionnaires d'événements***. Les gestionnaires d'événements sont joints à l'aide de l'opérateur `+=` et supprimés à l'aide de l'opérateur `-=`. L'exemple suivant joint un gestionnaire d'événements à l'événement `Changed` d'un `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

Pour les scénarios avancés où le contrôle du stockage sous-jacent d'un événement est souhaité, une déclaration d'événement peut fournir explicitement des accesseurs `add` et `remove`, qui sont plutôt similaires à l'accesseur `set` d'une propriété.

Opérateurs

Un ***opérateur*** est un membre qui définit la signification de l'application d'un opérateur d'expression particulière aux instances d'une classe. Trois types d'opérateurs peuvent être définis : les opérateurs unaires, les opérateurs binaires et les opérateurs de conversion. Tous les opérateurs doivent être déclarés comme `public` et `static`.

La classe `List<T>` déclare deux opérateurs, `operator==` et `operator!=`, et donne donc une nouvelle signification aux expressions qui appliquent ces opérateurs aux instances `List`. Plus précisément, les opérateurs définissent `List<T>` l'égalité de deux instances en comparant chacun des

`Equals` objets contenus à l'aide de leurs méthodes. L'exemple suivant utilise l'opérateur `==` pour comparer deux instances de `List<int>`.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

La première `Console.WriteLine` génère `True`, car les deux listes contiennent le même nombre d'objets avec les mêmes valeurs dans le même ordre. Si `List<T>` n'avait pas défini `operator==`, la première `Console.WriteLine` aurait affiché `False`, car `a` et `b` référencent des instances `List<int>` différentes.

Destructeurs

Un ***destructeur*** est un membre qui implémente les actions requises pour détruire une instance d'une classe. Les destructeurs ne peuvent pas avoir de paramètres, ils ne peuvent pas avoir de modificateurs d'accessibilité et ils ne peuvent pas être appelés explicitement. Le destructeur d'une instance est appelé automatiquement pendant garbage collection.

Le garbage collector est autorisé à atteindre une grande latitude en déterminant quand collecter des objets et exécuter des destructeurs. Plus précisément, le minutage des appels du destructeur n'est pas déterministe et les destructeurs peuvent être exécutés sur n'importe quel thread. Pour ces raisons, les classes doivent implémenter des destructeurs uniquement quand aucune autre solution n'est possible.

L'instruction `using` fournit une meilleure approche pour la destruction d'objets.

Structs

Comme les classes, les ***structs*** sont des structures de données qui peuvent contenir des membres de données et des fonctions membres. Mais contrairement aux classes, les structures sont des types valeur et ne nécessitent pas d'allocation de tas. Une variable de type struct stocke directement les données de la structure, alors qu'une variable de type class stocke une référence à un objet alloué dynamiquement. Les types struct ne prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object`.

Les structures sont particulièrement utiles pour les petites structures de données qui ont une sémantique par rapport à leurs valeurs. Les nombres complexes, les points dans un système de coordonnées ou les paires clé-

valeur dans un dictionnaire sont de bons exemples de structures. L'utilisation de structures plutôt que de classes pour les petites structures de données peut faire une grande différence dans le nombre d'allocations de mémoire effectuées par une application. Par exemple, le programme suivant crée et initialise un tableau de 100 points. Avec `Point` implémenté en tant que classe, 101 objets séparés sont instanciés : un pour le tableau et un pour chacun des 100 éléments.

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

Une alternative consiste à créer `Point` un struct.

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Maintenant, un seul objet est instancié, celui du tableau, et les instances `Point` sont stockées à la suite dans le tableau.

Vous appelez les constructeurs de struct avec l'opérateur `new`, mais cela n'implique pas que de la mémoire soit allouée. Au lieu d'allouer dynamiquement un objet et de renvoyer une référence à cet objet, un constructeur de struct retourne simplement la valeur du struct (généralement dans un emplacement temporaire sur la pile), et cette valeur est ensuite copiée si nécessaire.

Avec les classes, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les structs, les variables disposent chacune de leur propre copie des données, et il n'est pas possible pour les opérations sur une d'affecter les autres. Par exemple, la sortie produite par le fragment de code suivant varie selon `Point` qu'il s'agit d'une classe ou d'un struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Si `Point` est une classe, la sortie est `20` due `a` à `b` et référence le même objet. Si `Point` est un struct, la sortie est `10` due au fait que `a` l' `b` assignation de à crée une copie de la valeur et que cette copie n'est pas affectée par `a.x` l'assignation suivante à.

L'exemple précédent illustre deux des limitations des structs. Tout d'abord, la copie d'un struct entier est généralement moins efficace que la copie d'une référence d'objet, aussi le passage de paramètres d'affectation et de valeur peut être plus coûteux avec les structures qu'avec les types référence. Ensuite, à l'exception des paramètres `ref` et `out`, il n'est pas possible de créer des références aux structures, ce qui rend leur utilisation impossible dans un certain nombre de situations.

Tableaux

Un ***tableau*** est une structure de données qui contient un certain nombre de variables qui sont accessibles par des indices calculés. Les variables contenues dans un tableau, également appelé ***éléments*** du tableau, sont tous du même type, et ce type est appelé ***type d'élément*** du tableau.

Les types tableau sont des types référence, et la déclaration d'une variable tableau réserve simplement un espace pour une référence à une instance de tableau. Les instances de tableau réelles sont créées dynamiquement au moment de l' `new` exécution à l'aide de l'opérateur. L' `new` opération spécifie la ***longueur*** de la nouvelle instance de tableau, qui est ensuite résolue pour la durée de vie de l'instance. Les indices des éléments d'un tableau vont de `0` à `Length - 1`. L'opérateur `new` initialise automatiquement les éléments d'un tableau à leur valeur par défaut, c'est-à-dire, par exemple, zéro pour tous les types numériques et `null` pour tous les types référence.

L'exemple suivant crée un tableau de `int` éléments, initialise le tableau et imprime le contenu du tableau.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

Cet exemple crée et utilise un ***tableau unidimensionnel***. C# prend également en charge les ***tableaux multidimensionnels***. Le nombre de

dimensions d'un type tableau, également appelé ***rang*** du type tableau, est de un plus le nombre de virgules entre les crochets du type tableau. L'exemple suivant alloue un tableau unidimensionnel, à deux dimensions et à trois dimensions.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Le tableau `a1` contient 10 éléments, le tableau `a2` en contient 50 (10 x 5) et le tableau `a3` en contient 100 (10 × 5 × 2).

Le type d'élément d'un tableau peut être de n'importe quel type, y compris un type tableau. Un tableau avec des éléments d'un type tableau est parfois appelé un ***tableau en escalier***, car les longueurs des tableaux d'éléments ne sont pas nécessairement pas les mêmes. L'exemple suivant alloue un tableau de tableaux de `int` :

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La première ligne crée un tableau avec trois éléments, chacun de type `int[]` et chacun avec une valeur initiale de `null`. Les lignes suivantes initialisent ensuite les trois éléments avec des références aux instances individuelles de tableau de longueur variable.

L' `new` opérateur permet de spécifier les valeurs initiales des éléments du tableau à l'aide d'un ***initialiseur de tableau***, qui est une liste d'expressions écrites entre `{` les délimiteurs et. `}` L'exemple suivant alloue et initialise un `int[]` avec trois éléments.

```
int[] a = new int[] {1, 2, 3};
```

Notez que la longueur du tableau est déduite du nombre d'expressions entre `{` et. `}` Les déclarations locales des champs et variables peuvent être abrégées davantage afin de ne pas avoir à redéclarer le type tableau.

```
int[] a = {1, 2, 3};
```

Les deux exemples précédents sont équivalents à ce qui suit :

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

Une ***interface*** définit un contrat qui peut être implémenté par des classes

et structures. Une interface peut contenir des méthodes, des propriétés, des événements et des indexeurs. Une interface ne fournit pas les implémentations des membres qu'elle définit, elle indique simplement les membres qui doivent être fournis par les classes ou les structs qui implémentent l'interface.

Les interfaces peuvent utiliser ***l'héritage multiple***. Dans l'exemple suivant, l'interface `IComboBox` hérite à la fois de `ITextBox` et `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Les classes et les structs peuvent implémenter plusieurs interfaces. Dans l'exemple suivant, la classe `EditBox` implémente à la fois `IControl` et `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

Lorsqu'une classe ou un struct implémente une interface spécifique, les instances de cette classe ou struct peuvent être converties implicitement en ce type d'interface. Exemple :

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Si une instance n'est pas connue pour implémenter une interface donnée de façon statique, des casts de type dynamiques peuvent être utilisés. Par exemple, les instructions suivantes utilisent des casts de type dynamique pour obtenir les `IControl` implémentations d'interface et `IDataBound` d'objet. Étant donné que le type réel de l' `EditBox` objet est, les casts sont correctement exécutés.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

Dans la classe `EditBox` précédente, la `Paint` méthode de l' `IControl` interface et la `Bind` méthode de l' `IDataBound` interface sont implémentées `public` à l'aide de membres. C#prend également en charge les ***implémentations de membres d'interface explicites***, à l'aide desquelles la classe ou le struct peut éviter de créer les membres `public`. Une implémentation de membre d'interface explicite est écrite à l'aide du nom de membre d'interface qualifié complet. Par exemple, la classe `EditBox` peut implémenter les méthodes `IControl.Paint` et `IDataBound.Bind` à l'aide des implémentations de membres d'interface explicites, comme suit.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Les membres d'interface explicites sont accessibles uniquement via le type interface. Par exemple, l'implémentation de `IControl.Paint` fournie par la classe `EditBox` précédente ne peut être appelée qu'en `IControl` convertissant d'abord `EditBox` la référence au type d'interface.

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

Enums

Un ***type enum*** est un type valeur distinct avec un ensemble de constantes nommées. L'exemple suivant déclare et utilise un type `Color` Enum nommé avec trois valeurs constantes `Green`, `Red`, et `Blue`.

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Chaque type enum a un type intégral correspondant appelé le ***type sous-jacent*** du type enum. Un type enum qui ne déclare pas explicitement un type sous-jacent a un type `int` sous-jacent de. Le format de stockage d'un type enum et la plage de valeurs possibles sont déterminés par son type sous-jacent. L'ensemble de valeurs qu'un type enum peut utiliser n'est pas limité par ses membres enum. En particulier, toute valeur du type sous-jacent d'une énumération peut être convertie en type enum et est une valeur valide distincte de ce type enum.

L'exemple suivant déclare un type Enum nommé `Alignment` avec un type sous-jacent de. `sbyte`

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

Comme indiqué dans l'exemple précédent, une déclaration de membre enum peut inclure une expression constante qui spécifie la valeur du membre. La valeur de constante pour chaque membre enum doit être comprise dans la plage du type sous-jacent de l'enum. Lorsqu'une déclaration de membre enum ne spécifie pas explicitement de valeur, le membre reçoit la valeur zéro (s'il s'agit du premier membre du type enum)

ou la valeur du membre enum qui précède textuel plus un.

Les valeurs enum peuvent être converties en valeurs intégrales et vice versa à l'aide de casts de type. Exemple :

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

La valeur par défaut d'un type enum est la valeur intégrale zéro convertie en type enum. Dans les cas où les variables sont automatiquement initialisées à une valeur par défaut, il s'agit de la valeur donnée aux variables de types ENUM. Pour que la valeur par défaut d'un type enum soit facilement disponible, le littéral `0` convertit implicitement en un type enum. Ce qui suit est donc autorisé.

```
Color c = 0;
```

Délégués

Un ***type délégué*** représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction dans d'autres langages, mais contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

L'exemple suivant déclare et utilise un type délégué nommé `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

Une instance du type délégué `Function` peut faire référence à toute méthode qui accepte un argument `double` et retourne une valeur `double`. La `Apply` méthode applique un donné `Function` aux éléments d'un `double[]`, en retournant `double[]` un avec les résultats. Dans la méthode `Main`, `Apply` sert à appliquer trois fonctions différentes à un `double[]`.

Un délégué peut faire référence à une méthode statique (comme `Square` ou `Math.Sin` dans l'exemple précédent) ou à une méthode d'instance (comme `m.Multiply` dans l'exemple précédent). Un délégué qui référence une méthode d'instance référence aussi un objet particulier, et quand la méthode d'instance est appelée via le délégué, cet objet devient `this` dans l'appel.

Les délégués peuvent également être créés à l'aide de fonctions anonymes, qui sont des « méthodes inline » créées à la volée. Les fonctions anonymes peuvent voir les variables locales des méthodes qui l'entourent. Ainsi, l'exemple de multiplicateur ci-dessus peut être écrit plus facilement `Multiplier` sans utiliser une classe :

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

Une propriété intéressante et utile d'un délégué est qu'il ne connaît pas la

classe de la méthode qu'il référence, et cela lui est égal. Tout ce qui importe est que la méthode référencée ait les mêmes paramètres et type de retour que le délégué.

Attributs

Les types, membres et autres entités d'un programme C# prennent en charge les modificateurs qui contrôlent certains aspects de leur comportement. Par exemple, l'accessibilité d'une méthode est contrôlée à l'aide des modificateurs `public`, `protected`, `internal` et `private`. C# généralise cette fonctionnalité pour que les types d'informations déclaratives définis par l'utilisateur puissent être associés aux entités de programme et récupérés au moment de l'exécution. Les programmes spécifient ces informations déclaratives supplémentaires en définissant et en utilisant les ***attributs***.

L'exemple suivant déclare un attribut `HelpAttribute` qui peut être placé sur des entités de programme pour fournir des liens vers leur documentation associée.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Toutes les classes d'attributs dérivent de la `System.Attribute` classe de base fournie par l' .NET Framework. Les attributs peuvent être appliqués en donnant leur nom, ainsi que les éventuels arguments, à l'intérieur de crochets juste avant la déclaration associée. Si le nom d'un attribut se `Attribute` termine par, cette partie du nom peut être omise lorsque l'attribut est référencé. Par exemple, l'attribut `HelpAttribute` peut être utilisé comme suit.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic =
"Display")]
    public void Display(string text) {}
}
```

Cet `HelpAttribute` exemple attache un à la `Widget` classe et un autre

`HelpAttribute` à la `Display` méthode dans la classe. Les constructeurs publics d'une classe d'attributs contrôlent les informations qui doivent être fournies lorsque l'attribut est joint à une entité de programme. Des informations supplémentaires peuvent être fournies en référençant les propriétés en lecture-écriture publiques de la classe d'attributs (comme la référence à la propriété `Topic`, précédemment).

L'exemple suivant montre comment récupérer des informations d'attribut pour une entité de programme donnée au moment de l'exécution à l'aide de la réflexion.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

Lorsqu'un attribut particulier est demandé par réflexion, le constructeur de la classe d'attributs est appelé avec les informations fournies dans la source du programme, et l'instance d'attribut qui en résulte est retournée. Si des informations supplémentaires ont été fournies via les propriétés, ces propriétés sont définies sur les valeurs données avant que le renvoi de l'instance de l'attribut.

Critères spéciaux pour C# 7

02/11/2020 • 30 minutes to read • Edit Online

Les extensions de critères spéciaux pour C# permettent de nombreux avantages des types de données algébriques et des critères spéciaux des langages fonctionnels, mais d'une façon qui s'intègre parfaitement au langage sous-jacent. Les fonctionnalités de base sont : les types d'enregistrements, qui sont des types dont la signification sémantique est décrite par la forme des données ; et les critères spéciaux, qui sont un nouveau formulaire d'expression qui permet une décomposition à plusieurs niveaux extrêmement concise de ces types de données. Les éléments de cette approche sont inspirés par les fonctionnalités associées dans les langages de programmation F # et Scala.

Expression is

L' `is` opérateur est étendu pour tester une expression par rapport à un *modèle*.

```
relational_expression
    : relational_expression 'is' pattern
    ;
```

Cette forme de *relational_expression* s'ajoute aux formulaires existants dans la spécification C#. Il s'agit d'une erreur de compilation si le *relational_expression* à gauche du `is` jeton ne désigne pas une valeur ou n'a pas de type.

Chaque *identificateur* du modèle introduit une nouvelle variable locale qui est *définitivement assignée* une fois que l' `is` opérateur a la `true` valeur (c'est-à-dire *assignée de manière définitive quand la valeur est true*).

> Remarque : il existe techniquement une ambiguïté entre le *type* d'un `is-expression` et *constant_pattern*, qui peut être une analyse valide d'un identificateur qualifié. Nous essayons de la lier en tant que type pour la compatibilité avec les versions précédentes du langage. en cas d'échec, nous allons le résoudre comme nous le faisons dans d'autres contextes, à la première chose trouvée (qui doit être une constante ou un type). Cette ambiguïté est uniquement présente sur le côté droit d'une `is` expression.

Modèles

Les modèles sont utilisés dans l' `is` opérateur et dans un *switch_Statement* pour exprimer la forme des données par rapport à laquelle les données entrantes doivent être comparées. Les modèles peuvent être récursifs afin que certaines parties des données puissent être comparées à des sous-modèles.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

declaration_pattern
    : type simple_designation
    ;

constant_pattern
    : shift_expression
    ;

var_pattern
    : 'var' simple_designation
    ;
```

> Remarque : il existe techniquement une ambiguïté entre le *type* d'un `is-expression` et *constant_pattern*, qui peut être une analyse valide d'un identificateur qualifié. Nous essayons de la lier en tant que type pour la compatibilité avec les versions précédentes du langage. en cas d'échec, nous allons le résoudre comme nous le faisons dans d'autres contextes, à la première chose trouvée (qui doit être une constante ou un type). Cette ambiguïté est uniquement présente sur le côté droit d'une `is` expression.

Modèle de déclaration

Le *declaration_pattern* teste qu'une expression est d'un type donné et la convertit en ce type si le test a la valeur. Si le *simple_designation* est un identificateur, il introduit une variable locale du type donné nommé par l'identificateur donné. Cette variable locale est *assignée définitivement* lorsque le résultat de l'opération de correspondance de modèle est true.

```
declaration_pattern
    : type simple_designation
    ;
```

La sémantique d'exécution de cette expression est qu'elle teste le type de Runtime de l'opérande de *relational_expression* gauche par rapport au *type* dans le modèle. S'il s'agit d'un type de Runtime (ou d'un sous-type), le résultat de `is operator` est `true` . Elle déclare une nouvelle variable locale nommée par l' *identificateur* auquel est affectée la valeur de l'opérande de gauche lorsque le résultat est `true` .

Certaines combinaisons de type statique du côté gauche et du type donné sont considérées comme incompatibles et entraînent une erreur au moment de la compilation. Une valeur de type static `E` est dite compatible avec le *modèle* `T` s'il existe une conversion d'identité, une conversion de référence implicite, une conversion boxing, une conversion de référence explicite ou une conversion unboxing de `E` en `T` . Il s'agit d'une erreur de compilation si une expression de type `E` n'est pas compatible avec le type de modèle dans un modèle de type auquel elle est associée.

> Remarque : en C# 7,1, nous étendons cela pour permettre une opération de critères spéciaux si le type d'entrée ou le type `T` est un type ouvert. Ce paragraphe est remplacé par ce qui suit :
>
> Certaines combinaisons de type statique du côté gauche et du type donné sont considérées comme incompatibles et entraînent une erreur au moment de la compilation. Une valeur de type statique `E` est dite *compatible* avec le type `T` s'il existe une conversion d'identité, une conversion de référence implicite, une conversion boxing, une conversion de référence explicite ou une conversion unboxing de `E` en `T` , **ou si** `E` **ou** `T` **est un type ouvert**. Il s'agit d'une erreur de compilation si une expression de type `E` n'est pas compatible avec le type de modèle dans un modèle de type auquel elle est associée.

Le modèle de déclaration est utile pour effectuer des tests de type au moment de l'exécution de types référence et remplace l'idiome

```
var v = expr as Type;
if (v != null) { // code using v }
```

Avec un peu plus concis

```
if (expr is Type v) { // code using v }
```

Il s'agit d'une erreur si le *type* est un type valeur Nullable.

Le modèle de déclaration peut être utilisé pour tester des valeurs de types Nullable : une valeur de type `Nullable<T>` (ou un boxed `T`) correspond à un modèle de type `T2 id` si la valeur n'est pas null et que le type de `T2` est `T` , ou un type de base ou une interface de `T` . Par exemple, dans le fragment de code

```
int? x = 3;
if (x is int v) { // code using v }
```

La condition de l' `if` instruction est `true` au moment de l'exécution et la variable `v` contient la valeur `3` du type `int` dans le bloc.

Modèle de constante

```
constant_pattern
    : shift_expression
    ;
```

Un modèle de constante teste la valeur d'une expression par rapport à une valeur constante. La constante peut être une expression constante, telle qu'un littéral, le nom d'une variable déclarée `const` , une constante d'énumération ou une `typeof` expression.

Si les deux *e* et *c* sont des types intégraux, le modèle est considéré comme mis en correspondance si le résultat de l'expression `e == c` est `true` .

Dans le cas contraire, le modèle est considéré comme une correspondance si `object.Equals(e, c)` retourne `true` . Dans ce cas, il s'agit d'une erreur au moment de la compilation si le type statique de *e* n'est pas *compatible* avec le type de la constante.

Modèle var

```
var_pattern
    : 'var' simple_designation
    ;
```

Une expression *e* correspond à un *var_pattern* toujours. En d'autres termes, une correspondance avec un *modèle var* est toujours réussie. Si le *simple_designation* est un identificateur, alors, au moment de l'exécution, la valeur de *e* est liée à une variable locale nouvellement introduite. Le type de la variable locale est le type statique de *e*.

Il s'agit d'une erreur si le nom est `var` lié à un type.

Instruction switch

L' `switch` instruction est étendue pour sélectionner pour l'exécution le premier bloc ayant un modèle associé qui

correspond à l' *expression de switch*.

```
switch_label
    : 'case' complex_pattern case_guard? ':'
    | 'case' constant_expression case_guard? ':'
    | 'default' ':'
    ;

case_guard
    : 'when' expression
    ;
```

L'ordre dans lequel les modèles sont mis en correspondance n'est pas défini. Un compilateur est autorisé à faire correspondre les modèles dans le désordre et à réutiliser les résultats des modèles déjà associés pour calculer le résultat de la correspondance d'autres modèles.

Si une *protection de casse* est présente, son expression est de type `bool` . Elle est évaluée comme une condition supplémentaire qui doit être satisfaite pour que le cas soit considéré comme respecté.

Il s'agit d'une erreur si un *switch_label* peut n'avoir aucun effet au moment de l'exécution, car son modèle est inclus dans les cas précédents. [TODO : nous devrions être plus précis concernant les techniques que le compilateur est tenu d'utiliser pour atteindre ce jugement.]

Une variable de modèle déclarée dans un *switch_label* est assignée de manière définitive dans son bloc cas si et uniquement si ce bloc de cas contient précisément une *switch_label*.

[TODO : nous devons spécifier quand un *bloc switch* est accessible.]

Étendue des variables de modèle

La portée d'une variable déclarée dans un modèle est la suivante :

- Si le modèle est une étiquette case, alors l'étendue de la variable est le *bloc case*.

Dans le cas contraire, la variable est déclarée dans une expression *is_pattern* , et sa portée est basée sur la construction qui englobe immédiatement l'expression contenant l'expression *is_pattern* comme suit :

- Si l'expression se trouve dans une expression lambda avec une expression, son étendue est le corps de l'expression lambda.
- Si l'expression se trouve dans une méthode ou une propriété à expression nulle, son étendue est le corps de la méthode ou de la propriété.
- Si l'expression est dans une `when` clause d'une `catch` clause, sa portée est cette `catch` clause.
- Si l'expression se trouve dans un *iteration_statement*, son étendue est simplement cette instruction.
- Sinon, si l'expression est dans une autre forme d'instruction, son étendue est l'étendue contenant l'instruction.

Dans le but de déterminer l'étendue, un *embedded_statement* est considéré comme étant dans sa propre étendue. Par exemple, la grammaire d'un *if_statement* est

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

Par conséquent, si l'instruction contrôlée d'un *if_statement* déclare une variable de modèle, sa portée est limitée à ce *embedded_statement*.

```
if (x) M(y is var z);
```

Dans ce cas, la portée de `z` est l'instruction incorporée `M(y is var z);` .

D'autres cas sont des erreurs pour d'autres raisons (par exemple, dans la valeur par défaut d'un paramètre ou un attribut, les deux étant une erreur parce que ces contextes nécessitent une expression constante).

En C# 7,3, nous avons ajouté les contextes suivants dans lesquels une variable de modèle peut être déclarée :

- Si l'expression se trouve dans un *initialiseur de constructeur*, sa portée est l' *initialiseur de constructeur* et le corps du constructeur.
- Si l'expression se trouve dans un initialiseur de champ, son étendue est le *equals_value_clause* dans lequel elle apparaît.
- Si l'expression se trouve dans une clause de requête spécifiée pour être convertie dans le corps d'une expression lambda, son étendue est uniquement cette expression.

Modifications apportées à la désambiguïsation syntaxique

Il existe des situations impliquant des génériques où la grammaire C# est ambiguë et la spécification du langage indique comment résoudre ces ambiguïtés :

ambiguïtés de la grammaire 7.6.5.2

Les productions pour le *nom simple* (§ 7.6.3) et l' *accès aux membres* (§ 7.6.5) peuvent donner lieu à des ambiguïtés dans la grammaire des expressions. Par exemple, l'instruction suivante :

```
F(G<A,B>(7));
```

peut être interprété comme un appel à `F` avec deux arguments, `G < A` et `B > (7)` . Elle peut également être interprétée comme un appel à `F` avec un argument, qui est un appel à une méthode générique `G` avec deux arguments de type et un argument normal.

Si une séquence de jetons peut être analysée (en contexte) en tant que *nom simple* (§ 7.6.3), *accès aux membres* (§ 7.6.5) ou *accès aux membres de pointeurs* (§ 18.5.2) se terminant par un *type-argument-List* (§ 4.4.1), le jeton qui suit immédiatement le jeton de fermeture `>` est examiné. S'il s'agit de l'un des

```
( ) ] } : ; , . ? == != | ^
```

le *type-argument-List* est conservé dans le cadre du *simple-name*, *member-Access* ou *pointer-Member-Access* et toute autre analyse possible de la séquence de jetons est ignorée. Dans le cas contraire, la *liste d'arguments de type* n'est pas considérée comme faisant partie du *simple-name*, *member-Access* ou > *pointer-Member-Access*, même s'il n'y a aucune autre analyse possible de la séquence de jetons. Notez que ces règles ne sont pas appliquées lors de l'analyse d'un *type-argument-List* dans un *espace de noms ou un nom de type* (§ 3,8). L'instruction

```
F(G<A,B>(7));
```

, conformément à cette règle, sera interprété comme un appel à `F` avec un argument, qui est un appel à une méthode générique `G` avec deux arguments de type et un argument normal. Les instructions

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

est interprété comme un appel à `F` avec deux arguments. L'instruction

```
x = F < A > +y;
```

est interprété comme un opérateur inférieur à, supérieur à et opérateur unaire plus, comme si l'instruction avait été écrite, et `x = (F < A) > (+y)` non comme un *simple-name* avec un *type-argument-List* suivi d'un opérateur binaire plus. Dans l'instruction

```
x = y is C<T> + z;
```

les jetons `C<T>` sont interprétés comme un *nom de type ou d'espace de noms* avec un *type-argument-List*.

Il y a un certain nombre de modifications introduites dans C# 7 qui rendent ces règles de désambiguïsation ne suffisent plus pour gérer la complexité du langage.

Déclarations de variables out

Il est désormais possible de déclarer une variable dans un argument out :

```
M(out Type name);
```

Toutefois, le type peut être générique :

```
M(out A<B> name);
```

Étant donné que la grammaire de la langue de l'argument utilise l' *expression*, ce contexte est soumis à la règle de désambiguïsation. Dans ce cas, la fermeture `>` est suivie d'un *identificateur*, qui n'est pas l'un des jetons qui lui permet d'être traité comme un *type-argument-List*. Je propose donc d' **Ajouter l' *identificateur* à l'ensemble de jetons qui déclenche la désambiguïsation pour une liste d'arguments de *type*.**

Déclarations de tuples et de déconstruction

Un littéral de tuple s'exécute exactement dans le même problème. Considérer l'expression de Tuple

```
(A < B, C > D, E < F, G > H)
```

Dans les anciennes règles C# 6 pour l'analyse d'une liste d'arguments, celle-ci est interprétée comme un tuple avec quatre éléments, à partir de `A < B` la première. Toutefois, lorsque cela apparaît à gauche d'une déconstruction, nous voulons que la désambiguation soit déclenchée par le jeton d' *identificateur* comme décrit ci-dessus :

```
(A<B,C> D, E<F,G> H) = e;
```

Il s'agit d'une déclaration de déconstruction qui déclare deux variables, la première étant de type `A<B,C>` et nommée `D` . En d'autres termes, le littéral de tuple contient deux expressions, chacune d'elles étant une expression de déclaration.

Pour simplifier la spécification et le compilateur, je propose que ce littéral de tuple soit analysé comme un tuple à deux éléments à chaque fois qu'il apparaît (qu'il apparaisse ou non sur le côté gauche d'une assignation). Il s'agit d'un résultat naturel de la désambiguïsation qui est décrite dans la section précédente.

Critères spéciaux

Les critères spéciaux présentent un nouveau contexte dans lequel l'ambiguïté de type expression se produit. Précédemment, le côté droit d'un `is` opérateur était un type. À présent, il peut s'agir d'un type ou d'une expression, et s'il s'agit d'un type, il peut être suivi d'un identificateur. Cela peut, techniquement, modifier la

signification du code existant :

```
var x = e is T < A > B;
```

Cela peut être analysé dans les règles C# 6 comme

```
var x = ((e is T) < A) > B;
```

mais sous les règles C# 7 (avec la désambiguïsation proposée ci-dessus) sont analysées comme

```
var x = e is T<A> B;
```

qui déclare une variable `B` de type `T<A>` . Heureusement, les compilateurs natif et Roslyn ont un bogue qui leur donne une erreur de syntaxe sur le code C# 6. Par conséquent, cette modification avec rupture particulière n'est pas un problème.

La mise en correspondance de modèle introduit des jetons supplémentaires qui doivent conduire la résolution d'ambiguïté à la sélection d'un type. Les exemples suivants de code C# 6 valide existant seraient rompus sans règles de désambiguïsation supplémentaires :

```
var x = e is A<B> && f;            // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;             // &
var x = e is A<B>[];               // [
```

Proposition de modification de la règle de désambiguïsation

Je propose de modifier la spécification pour modifier la liste des jetons disambiguating à partir de

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

Et, dans certains contextes, nous traitons l' *identificateur* en tant que jeton disambiguating. Ces contextes sont l'endroit où la séquence de jetons enlevée est immédiatement précédée de l'un des mots clés `is` , `case` , ou `out` , ou se présente lors de l'analyse du premier élément d'un littéral de Tuple (auquel cas les jetons sont précédés de `(` ou `:` et l'identificateur est suivi d'un `,`) ou d'un élément suivant d'un littéral de Tuple.

Règle de désambiguïté modifiée

La règle de désambiguïté révisée serait semblable à celle-ci

> Si une séquence de jetons peut être analysée (en contexte) en tant que *nom simple* (§ 7.6.3), *accès aux membres* (§ 7.6.5) ou *accès aux membres de pointeurs* (§ 18.5.2) se terminant par un *type-argument-List* (§ 4.4.1), le jeton qui suit immédiatement le jeton de fermeture `>` est examiné pour voir s'il est
>
> - Un de `( ) ] } : ; , . ? == != | ^ && || & [` ; ou
> - Un des opérateurs relationnels `< > <= >= is as` ; ou
> - Mot clé de requête contextuelle apparaissant à l'intérieur d'une expression de requête ; ni

- Dans certains contextes, nous traitons l' *identificateur* en tant que jeton disambiguating. Ces contextes sont là où la séquence de jetons enlevée est immédiatement précédée de l'un des mots clés `is` , `case` ou, ou se présente lors de `out` l'analyse du premier élément d'un littéral de Tuple (auquel cas les jetons sont précédés de `(` ou `:` et l'identificateur est suivi d'un `,`) ou d'un élément suivant d'un littéral de Tuple.

Si le jeton suivant figure parmi cette liste, ou un identificateur dans un tel contexte, le *type-argument-List* est conservé dans le cadre du *simple-name*, de l' *accès aux membres* ou du *pointeur-Member Access* , et toute autre analyse possible de la séquence de jetons est ignorée. Dans le cas contraire, la *liste d'arguments de type* n'est pas considérée comme faisant partie du *simple-name*, d' *accès aux membres* ou d' *accès aux membres du pointeur*, même s'il n'y a pas d'autre analyse possible de la séquence de jetons. Notez que ces règles ne sont pas appliquées lors de l'analyse d'un *type-argument-List* dans un *espace de noms ou un nom de type* (§ 3,8).

Modifications avec rupture en raison de cette proposition

Aucune modification avec rupture n'est connue en raison de cette règle de désambiguïsation d'ambiguïté proposée.

Exemples intéressants

Voici quelques résultats intéressants de ces règles de désambiguïsation :

L'expression `(A < B, C > D)` est un tuple avec deux éléments, chacun étant une comparaison.

L'expression `(A<B,C> D, E)` est un tuple avec deux éléments, le premier étant une expression de déclaration.

L'appel `M(A < B, C > D, E)` a trois arguments.

L'appel `M(out A<B,C> D, E)` a deux arguments, le premier étant une `out` déclaration.

L'expression `e is A<B> C` utilise une expression de déclaration.

L'étiquette case `case A<B> C:` utilise une expression de déclaration.

Exemples de critères spéciaux

Est de

Nous pouvons remplacer l'idiome

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

Avec un peu plus concis et direct

```
if (expr is Type v) {
    // code using v
}
```

Tester la valeur null

Nous pouvons remplacer l'idiome

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

Avec un peu plus concis et direct

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Simplification arithmétique

Supposons que nous définissons un ensemble de types récursifs pour représenter des expressions (selon une proposition distincte) :

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Nous pouvons maintenant définir une fonction pour calculer la dérivée (non réduite) d'une expression :

```
Expr Deriv(Expr e)
{
  switch (e) {
    case X(): return Const(1);
    case Const(*): return Const(0);
    case Add(var Left, var Right):
      return Add(Deriv(Left), Deriv(Right));
    case Mult(var Left, var Right):
      return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
    case Neg(var Value):
      return Neg(Deriv(Value));
  }
}
```

Une expression simplifie les modèles positionnels :

```
Expr Simplify(Expr e)
{
  switch (e) {
    case Mult(Const(0), *): return Const(0);
    case Mult(*, Const(0)): return Const(0);
    case Mult(Const(1), var x): return Simplify(x);
    case Mult(var x, Const(1)): return Simplify(x);
    case Mult(Const(var l), Const(var r)): return Const(l*r);
    case Add(Const(0), var x): return Simplify(x);
    case Add(var x, Const(0)): return Simplify(x);
    case Add(Const(var l), Const(var r)): return Const(l+r);
    case Neg(Const(var k)): return Const(-k);
    default: return e;
  }
}
```

Procédures pas à pas C#

02/11/2020 • 4 minutes to read • Edit Online

Les procédures pas à pas fournissent des instructions détaillées pour les scénarios courants, ce qui en fait un bon point de départ pour apprendre à utiliser le produit ou une fonctionnalité particulière.

Cette section contient des liens vers les procédures pas à pas de la programmation en C#.

Contenu de cette section

- Traiter les tâches asynchrones à mesure qu'elles se terminent
  Explique comment créer une solution asynchrone avec async et await.
- Création d'un composant Windows Runtime en C# ou Visual Basic et appel de ce composant à partir de JavaScript
  Montre comment créer un type de Windows Runtime, le Packager dans un composant Windows Runtime, puis appeler le composant à partir d'une application Windows 8. x Store générée pour Windows à l'aide de JavaScript.
- Programmation Office (C# et Visual Basic)
  Montre comment créer un classeur Excel et un document Word à l'aide de C# et de Visual Basic.
- Création et utilisation d'objets dynamiques (C# et Visual Basic)
  Montre comment créer un objet personnalisé qui expose dynamiquement le contenu d'un fichier texte et comment créer un projet qui utilise la bibliothèque `IronPython`.
- Création d'un contrôle composite à l'aide de Visual C #
  Montre comment créer un contrôle composite simple et étendre ses fonctionnalités en se servant de l'héritage.
- Création d'un contrôle de Windows Forms qui tire parti des fonctionnalités au moment du design de Visual Studio
  Montre comment créer un concepteur personnalisé pour un contrôle personnalisé.
- Héritage d'un contrôle Windows Forms à l'aide de Visual C #
  Montre comment créer un contrôle bouton hérité simple. Ce bouton hérite des fonctionnalités du bouton Windows Forms standard et expose un membre personnalisé.
- Débogage de contrôles Windows Forms personnalisés au moment du design
  Décrit comment déboguer le comportement au moment du design de votre contrôle personnalisé.
- Procédure pas à pas : effectuer des tâches courantes à l'aide d'actions de concepteur
  Présente quelques tâches couramment exécutées, notamment l'ajout ou la suppression d'un onglet sur un `TabControl`, l'ancrage d'un contrôle à son parent et la modification de l'orientation d'un contrôle `SplitContainer`.
- Écriture de requêtes en C# (LINQ)
  Montre les fonctionnalités du langage C# utilisées pour écrire des expressions de requête LINQ.
- Manipulation de données (C#) (LINQ to SQL)
  Décrit un scénario LINQ to SQL pour l'ajout, la modification et la suppression des données dans une base de données.
- Requête et modèle objet simples (C#) (LINQ to SQL)

Montre comment créer une classe d'entité et une requête simple pour filtrer la classe d'entité.

- Utilisation de procédures stockées uniquement (C#) (LINQ to SQL)
  Montre comment utiliser LINQ to SQL pour accéder aux données en exécutant uniquement des procédures stockées.

- Interrogation de relations (C#) (LINQ to SQL)
  Montre comment utiliser des associations LINQ to SQL pour représenter les relations de clé étrangère dans une base de données.

- Écriture d'un visualiseur en C #
  Montre comment écrire un visualiseur simple à l'aide de C#.

Sections connexes

- Exemples et procédures pas à pas relatifs au déploiement
  Fournit des exemples pas à pas de scénarios de déploiement courants.

Voir aussi

- Guide de programmation C#
- Visual Studio Samples